Graduado en Ingeniería Informática

FACULTAD DE INFORMÁTICA

UNIVERSIDAD POLITÉCNICA DE MADRID

TRABAJO FIN DE GRADO

SIMULACIÓN DE EVOLUCIÓN DIRIGIDA DE BACTERIOFAGOS EN POBLACIONES DE BACTERIAS EN 2D

AUTOR: CÉSAR PUMAR GARCÍA

TUTOR: ALFONSO RODRÍGUEZ-PATÓN ARADAS

MADRID, JUNIO DE 2013

Summary

The research group is currently developing a biological computing model to be implemented with Escherichia Coli bacteria and bacteriophages M13, but it has to be modelled and simulated before any experiment in order to reduce the amount of failed attempts, time and costs.

The problem that gave rise to this project is that there are no software tools which are able to simulate the biological process underlying that computational model, so it needs to be developed before doing any experimental implementation. There are several software tools which can simulate most of the biological processes and bacterial interactions in which this model is based, so what needs to be done is to study those available simulation tools, compare them and choose the most appropriate in order to be improved adding the desired functionality for this design.

Directed evolution is a method used in biotechnology to obtain proteins or nucleic acids with properties not found in nature. It consists of three steps: 1) creating a library of mutants, 2) selecting the mutants with the desired properties, 3) replicating the variants identified in the selection step. The new software tool will be verified by simulating the selection step of a process of directed evolution applied to bacteriophages.

Resumen

El grupo de investigación está desarrollando un modelo de computación biológica para ser implementado con bacterias Escherichia Coli y bacteriofagos M13, aunque primero tiene que ser modelizado antes de realizar cualquier experimento, de forma que los intentos fallidos y por lo tanto los costes se verán reducidos.

El problema que dio lugar a este proyecto es la ausencia de herramientas software capaces de simular el proceso biológico que subyace a este modelo de computación biológica, por lo que dicha herramienta tiene que ser desarrollada antes de realizar cualquier implementación real. Existen varias herramientas software capaces de simular la mayoría de los procesos biológicos y las interacciones entre bacterias en los que se basa este modelo, por lo que este trabajo consiste en realizar un estudio de dichas herramientas de simulación, compararlas y escoger aquella más apropiada para ser mejorada añadiendo la funcionalidad deseada para este diseño.

La evolución dirigida es un método utilizado en biotecnología para obtener proteínas o ácidos nucleicos con propiedades que no se encuentran en la naturaleza. Este método consiste en tres pasos: 1) crear una librería de mutantes, 2) seleccionar los mutantes con las propiedades deseadas, 3) Replicar los mutantes deseados. La nueva herramienta software será verificada mediante la simulación de la selección de mutantes de un proceso de evolución dirigida aplicado a bacteriofagos.

Contents

Chapter 1

Introduction

1.1 Document Structure

Introduction: This section describes Bacterial Computing as a subfield of Natural Computing and Synthetic Biology. It then explains the need for this project.

State of the art: It starts describing the basic molecules involved in the cellular machinery, how a cell works using those molecules and how to program a cell from a computer scientist point of view. It then explains how basic computations work into a cell using the gene regulatory networks, and how the cells can perform algorithmic communications using the diffusion of molecules in the media, sexual reproduction as a wired channel and bacteriophage viruses as a wireless channel. It describes the main features, the infection process and the phage display technology using the M13 bacteriophage. It finally describes how to model bacterial populations and what the main simulation tools for this purpose are.

Analysis of BactoSim I: In the first place, an overview of the software from an user point of view is explained here, describing the main features of the tool. Secondly, a depther analysis of the software is made, using diagrams and modelling languages to explain and represent its modules.

System design and implementation: Using the data obtained from the analysis in the previous section, it is proposed here what changes in the software are needed in order to make it able to simulate the infection mechanism using M13 bacteriophages.

System evaluation: All the changes in the software are tested. When the software works properly, a model fitting and experimental validation are performed in order to check if the simulation matches the experimental results.

Finally, there is a section for conclusions and future work, another one for the references and the last one for the appendices.

1.2 Bacterial Computing

When people listen to the word "computer" they think automatically in CPUs, screens, mouses and keyboards, but there are many types of computers beyond personal computers, like supercomputers or embedded systems, which can be found everywhere. Thinking more widely, a computer is any device which can perform computations, it does not matter what the technology underlying that device is, or what kind of computations it performs. The only that matters is that it computes. Computers have not always been based on electronics. In fact, the first computers where mechanic. Why could not they be based on another technology?

There is a sub-field of Computer Science called Natural Computing. It has four main goals: 1) Studying Nature to develop new bio-inspired algorithms. 2) Understanding the properties of living organisms in order to implement Artificial Life with electronic computers or other devices. 3) Using Nature as substrate to implement computations, which leads to Quantum Computing or Molecular Computing. 4) Looking for natural processes which may be observed as information processing.

There is a lot of research in the fields of Quantum and Molecular computing, which seeks to use quanta or molecules respectively as technologies in order to develop new computers which can replace the electronic ones. But there is also an increasingly interest in the fourth goal, mainly in the observation of biological processes as information processing, which could lead to the development of new computers based in biological technology. Information processing in biological substrates is slower in terms of computing, so these computers would not be able to replace the electronic ones, but they could perform different tasks that an electronic computer would not be able to do. Those tasks could be processing molecules or biological signals into a living organism instead of electrical 0's or 1's taken from an electronic device as inputs, and returning other molecules or signals as outputs.

This project is focused on the last goal using biological processes found in bacteria, so it is called Bacterial Computing (figure 1.1[1]). It is also a subfield of Synthetic Biology, due to Bacterial Computing is based on it in order to synthesize new bacteria whose behaviours are the implementation of algorithms, communication protocols or digital circuits.

[1]Source: Andrianantoandro, Ernesto, et al. "Synthetic biology: new engineering rules for an emerging discipline." Molecular systems biology 2.1 (2006). Permission obtained from Nature Publishing Group

Bacteria may be programmed using Synthetic Biology techniques, like building a circular DNA molecule called plasmid using Biobricks and inserting it into a bacterium. These Biobricks are sets of genes with a defined behaviour, and they can be found in a well know database for Biobricks, which will be explained in its corresponding section. When the plasmid is inserted into a bacterium, such bacterium behaves in the way it is programmed.

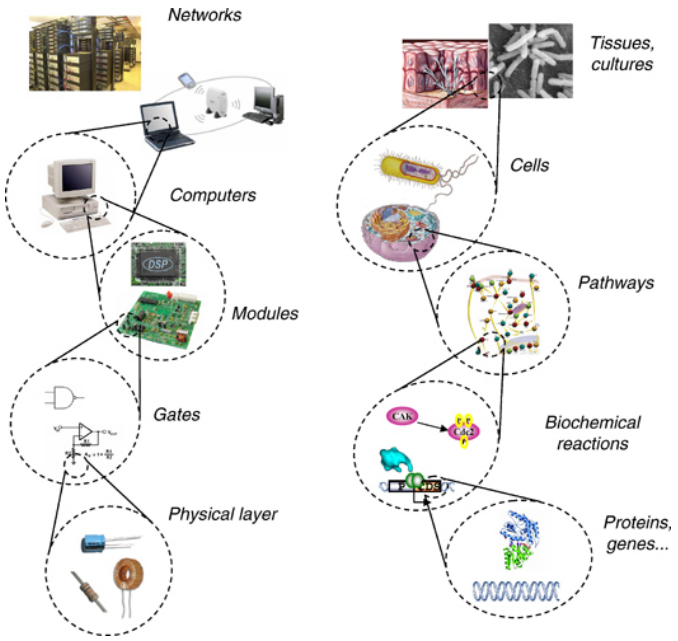


FIGURE 1.1: **Computer Engineering vs Synthetic Biology**

Bacterial populations are ideal to perform distributed, parallel computations. They have two ways to communicate: Quorum Sensing, which consists on the diffusion of small molecules in the media, or DNA messaging via sexual reproduction or viral infection. This project is focused in DNA messaging based on viral infection.

1.3 Purpose of this project

Synthetic Biology is considered science and engineering, so it is based on several engineering processes:

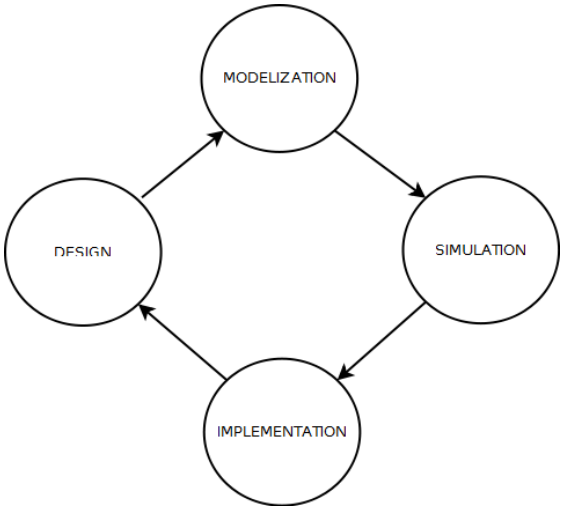


FIGURE 1.2: Main processes involved in synthetic biology

The research group is currently developing a computational model to be implemented with Escherichia Coli bacteria and bacteriophages M13, but it has to be modelled and simulated before any experiment in order to reduce the amount of failed attempts, time and costs.

The problem that gave rise to this project is that there are no software tools which are able to simulate the biological process underlying that computational model, so it needs to be developed before doing any experimental implementation. There are several software tools which can simulate most of the biological processes and bacterial interactions in which this model is based, so what needs to be done is to study those available simulation tools, compare them and choose the most appropriate in order to be improved adding the desired functionality for this design.

Directed evolution is a method used in biotechnology to obtain proteins or nucleic acids with properties not found in nature. It consists of 1) creating a library of mutants. 2) selecting the mutants with the desired properties 3) replicating the variants identified in the selection. The new software tool will be verified by simulating the selection step of a directed evolution process applied to bacteriophages.

Chapter 2

State of the Art

2.1 Fundamentals of Molecular Biology

In this section, the basic molecules and machinery needed to understand how a cell works are explained. Those molecules are nucleic acids (DNA and RNA), enzymes which modify nucleic acids, and proteins. The Fundamental Dogma of Molecular Biology, in which all the cellular processes are based is also explained, including transcription, translation and ribosomes.

2.1.1 The building blocks: Nucleic Acids

Deoxyribonucleic acid (DNA) is a biological macromolecule formed by a sequence of four nitrogenous bases, named Adenine, Guanine, Cytosine and Thymine (hereafter A, G, C and T). That sequence may be any combination of variable length of those nitrogenous bases. This macromolecule encodes the genetic instructions underlying all the biological processes which can be found into the cells. Formally speaking, DNA molecules are words of variable length formed by the alphabet $\Sigma_{DNA} = \{A, G, C, T\}$.

Those nitrogenous bases are complementary between them, as shown in figure 2.1. Adenine matches with Cytosine, and Guanine matches with Thymine. What does it mean? It means that a single DNA strand (ssDNA) may be coupled to its complementary strand, forming a double stranded DNA (dsDNA) molecule.

Hence, a DNA molecule X has a complementary molecule $\overline{X}$:

$X = AGCCT \longrightarrow \overline{X} = CTAAG$

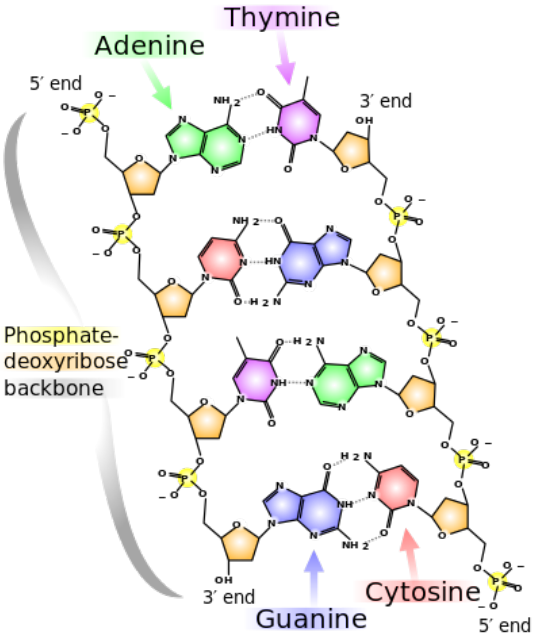


FIGURE 2.1: **Nitrogenous bases**

Ribonucleic acid (RNA) is another type of macromolecule similar to DNA, but they mainly differ in one of the nitrogenous bases which composes them. The nitrogenous base that RNA uses instead of Thymine is Uracyl (U), so the alphabet by which RNA words are formed is $\Sigma_{RNA} = \{A, G, C, U\}$. In this case, G and T are complementary, so an RNA molecule may also have a complementary strand:

$$X = AGCC\mathbf{U} \longrightarrow \overline{X} = CTAA\mathbf{G}$$

Both DNA and RNA are modified by several molecules named enzymes, which are explained in the following section.

2.1.2 Enzymes involved in Genetics

RNA molecules may be obtained from DNA, and the opposite process is also possible. Double-stranded DNA molecules may be obtained from single-stranded DNA, and those molecules may also serve as template to create new copies of them. Two molecules may be concatenated, and a single molecule may also be split. How are all those operations possible? There are several biomolecules named enzymes, which can be formally represented as operators over DNA and RNA words. The following explanations are simplifications and abstractions of the overall processes involved in these operations:

- DNA Helicase (dsDNA to ssDNA): This enzyme splits a double-stranded DNA molecule in two single-stranded DNA molecules (the original and the complementary).

- DNA Polymerases (DNA Complement): When these types of enzymes bind to a DNA single-stranded molecule, they create the complement of the original strand, which remains bound to the original one, resulting in a double-stranded DNA molecule. - RNA Polymerases (DNA to RNA): This molecule binds to an existing DNA molecule and uses it as a template to create its corresponding RNA complementary strand. - DNA Ligases (Concatenation): It allows joining two DNA strands.

- Nucleases (split): They are enzymes which can cut a DNA strand, forming two DNA molecules.

There are more enzymes involved in the cellular processes, but those are the most important for the understanding of the basic biological concepts underlying this project. DNA and RNA polymerases will be used again in the following chapters.

2.1.3 Proteins

This molecules are the building blocks of many biological structures and they play an important role in most of the genetic processes. They are sequences of aminoacids (organic molecules made from amine and carboxylic acid functional groups). Those aminoacids are obtained from a biological process called translation, which will be explained in the following section. It is anticipated that three base pairs from RNA (also known as triplet codons) are translated into one of the 22 standard aminoacids, as the following table shows:

1st base

2nd base	U		C		A		G		3rd base
U	UUU	Phenylalanine	UCU	Serine	UAU	Tyrosine	UGU	Cysteine	U
	UUC	Phenylalanine	UCC	Serine	UAC	Tyrosine	UGC	Cysteine	C
	UUA	Leucine	UCA	Serine	UAA	Stop	UGA	Stop	A
	UUG	Leucine	UCG	Serine	UAG	Stop	UGG	Tryptophan	G
C	CUU	Leucine	CCU	Proline	CAU	Histidine	CGU	Arginine	U
	CUC	Leucine	CCC	Proline	CAC	Histidine	CGC	Arginine	C
	CUA	Leucine	CCA	Proline	CAA	Glutamine	CGA	Arginine	A
	CUG	Leucine	CCG	Proline	CAG	Glutamine	CGG	Arginine	G
A	AUU	Isoleucine	ACU	Threonine	AAU	Asparagine	AGU	Serine	U
	AUC	Isoleucine	ACC	Threonine	AAC	Asparagine	AGC	Serine	C
	AUA	Isoleucine	ACA	Threonine	AAA	Lysine	AGA	Arginine	A
	AUG	Methionine (Start)	ACG	Threonine	AAG	Lysine	AGG	Arginine	G
G	GUU	Valine	GCU	Alanine	GAU	Aspartic Acid	GGU	Glycine	U
	GUC	Valine	GCC	Alanine	GAC	Aspartic Acid	GGC	Glycine	C
	GUA	Valine	GCA	Alanine	GAA	Glutamic Acid	GGA	Glycine	A
	GUG	Valine	GCG	Alanine	GAG	Glutamic Acid	GGG	Glycine	G

FIGURE 2.2: **Triplet codon table**

2.1.4 Central Dogma of Molecular Biology

This section explains how the genetic machinery works into cells. There are three main processes involved: DNA replication, DNA transcription and RNA translation:

- The first process is accomplished by DNA Helicase and Polymerase among others, as explained in the previous sections.

- DNA transcription consists on the copy of a DNA into a RNA, carried out by RNA polymerase which creates a complementary, antiparallel RNA strand.

- RNA translation is accomplished by a macromolecule called ribosome, which can be formally represented as a device that reads RNA sequences as input and creates aminoacid sequences (proteins) as output. As it was explained in the previous section, three RNA bases correspond to one aminoacid.

The overall process is known as the Central Dogma of Molecular Biology and all living organisms are based on it. This processes are used in the following sections as the basis of Bacterial Computing.

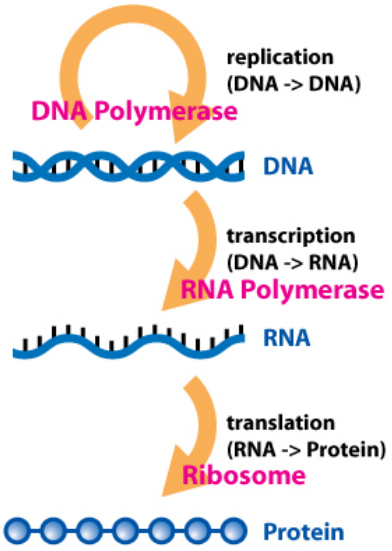


FIGURE 2.3: **Central Dogma of Molecular Biology**

2.2 How to build a biocomputer

In this section, the main characteristics of cells are explained. Then, plasmids are indicated as the programs which are inserted into bacteria, and it is shown how to code and run those programs.

2.2.1 Basic machinery: cells

Cells are living organisms that exist thanks to a phospholipid membrane which allows them to be isolated from the external aqueous media but at the same time it grants them an internal aqueous media (cytoplasm). This is the internal media where all cellular processes take place. Furthermore, they can exchange substances with the external media through their membrane (therefore with other cells).

There are two types of cells: Prokaryotes (archaea and bacteria) and eukaryotes (animal and vegetable cells, Fungi and protists). Prokaryotes do not have any cellular nucleous, i.e., their genetic material is dispersed in the cytoplasm, mainly in the nucleoid. Eukaryotes have their genetic material gathered into a lypid bilayer called nucleous. This project is based on Escherichia Coli bacteria, so that the explanation of prokaryotic cells is extended below.

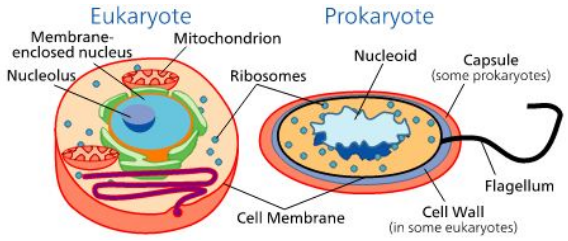


FIGURE 2.4: **Eukaryotes vs Prokaryotes**

There is a classification for bacteria which consists of how their cellular wall reacts with Gram staining. Some bacteria are visualized as purple (Gram-positive), and those which turn pink are called Gram-negative. Gram-positive have a gross cellular wall, while Gram-negative have a thinner one. It is anticipated that Escherichia Coli is a Gram-negative bacterium.

It was already explained that bacteria contain most of their genetic material in the nucleoid, but it may also be found as sparse plasmids (circular, double-stranded DNA molecules) in the cytoplasm although these plasmids may be incorporated in the nucleoid.

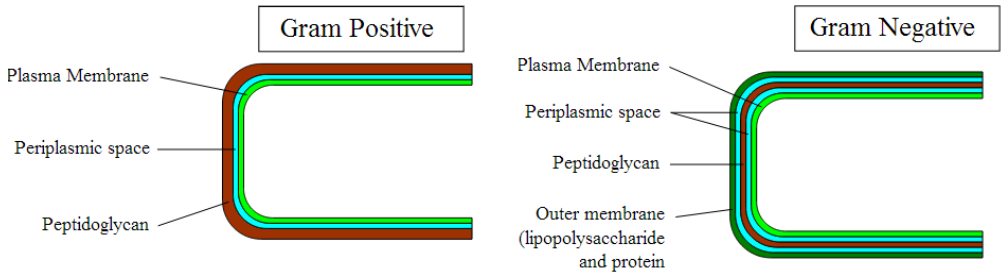


FIGURE 2.5: **Gram-positive vs Gram-negative**

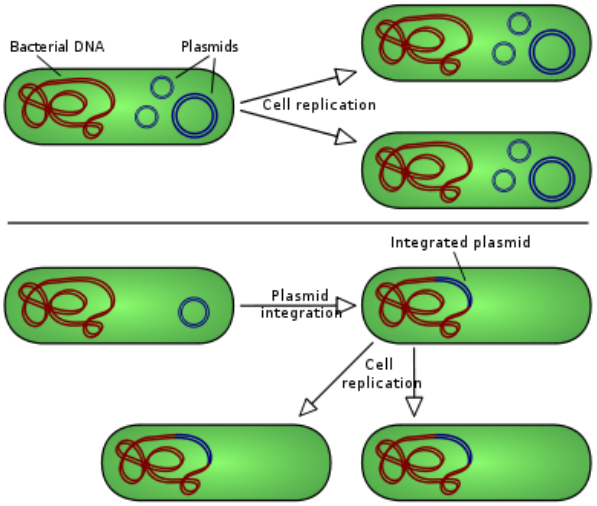


FIGURE 2.6: **Plasmids**

There are some Genetic Engineering techniques to synthesize plasmids containing the desired genes in the laboratory, so that they can be incorporated into bacteria (plasmid vectors). This is the method used to program a bacterium, explained in the next section.

2.2.2 Developing a bioprogram

Bacterial DNA is the main substrate to implement algorithms or digital circuits. Biologists synthesize a circular DNA strand called plasmid, which contains the chosen genes in order to the bacteria to implement the desired computation, and they insert it into the bacterium. The bacterium reads the genes placed in that plasmid and runs the instructions represented by those genes [1].

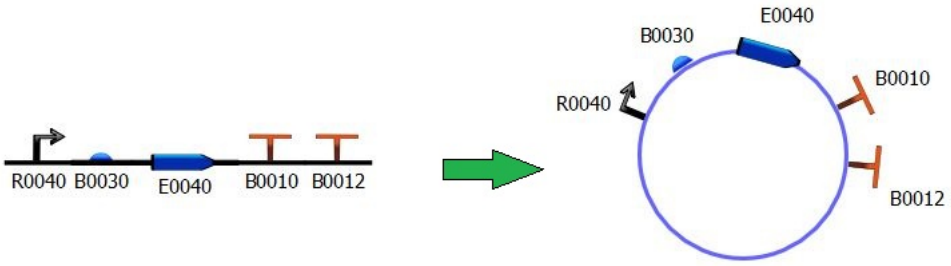


FIGURE 2.7: **Design and implementation of a biological bioprogram using plasmids**

That is to say, a bio-programmer who builds a bio-program (plasmid) would have to choose what genes should be included on it, and how they would be sorted. There is a database containing information about the genes available for that purpose (http://partsregistry.org). These genes are considered the building blocks, that is the reason why they are called Biobricks (http://biobricks.org).

2.2.3 Running a bioprogram

How is a bacterium able to read and run the inserted bioprogram? Following the Central Dogma of Molecular Biology, the plasmid DNA is transcripted into RNA by RNA Polymerase, which is translated into proteins by ribosomes.

What does the cell with those proteins obtained from the plasmid DNA? Cells can be considered machines whose behaviour is based on the interactions between their proteins and genes (and other substances which are not contained in the scope of this project), therefore their behaviour is determined by the kind of proteins contained on them and how those proteins interact between them and their genes. The DNA segments which lead to those interactions are called Gene Regulatory Networks. In fact, this name refers to both natural and synthetic DNA contained into a bacterium, so that plasmid DNA can also interact with the natural DNA of the cell (this fact may lead to undesired interactions, and that is the reason why there are actually some research projects about how to minimize the genome of cells).

Different interactions between proteins and genes can be obtained by modifying the bacterial genome, so that the desired behaviours can be achieved. Some examples of these behaviours (which can be computations) are explained in the following section.

2.3 Computing with Gene Regulatory Networks

There are several steps in the protein synthesis from genes. As it was explained before, the first step is transcription and the second one is translation. But proteins may modify their structure by folding upon themselves or interacting with other proteins. Those interactions take place in another process called post-translation, and Synthetic Biology (so does Bacterial Computing) modules can be implemented with each of those three processes [2]. To simplify, only the first process (computing by transcriptional control) is explained here.

When the fact that a bioprogram consists on the interactions between proteins and genes is understood, it remains only to explain how those interactions work, so two examples of logic gates [3] (NOT and AND, respectively) are proposed[2]:

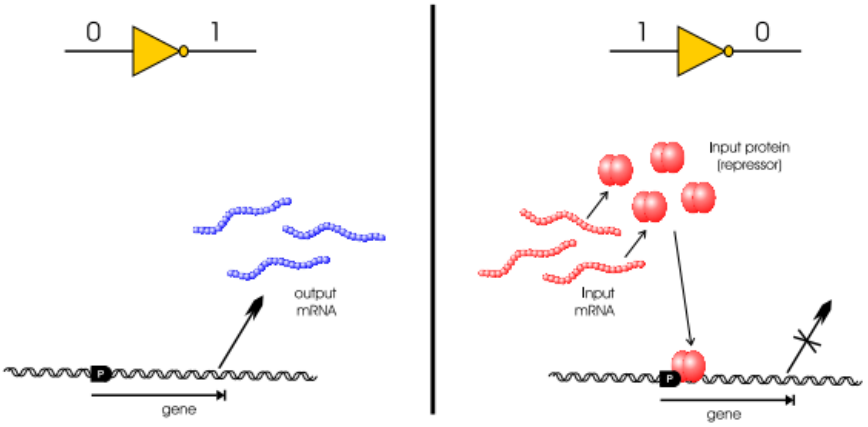
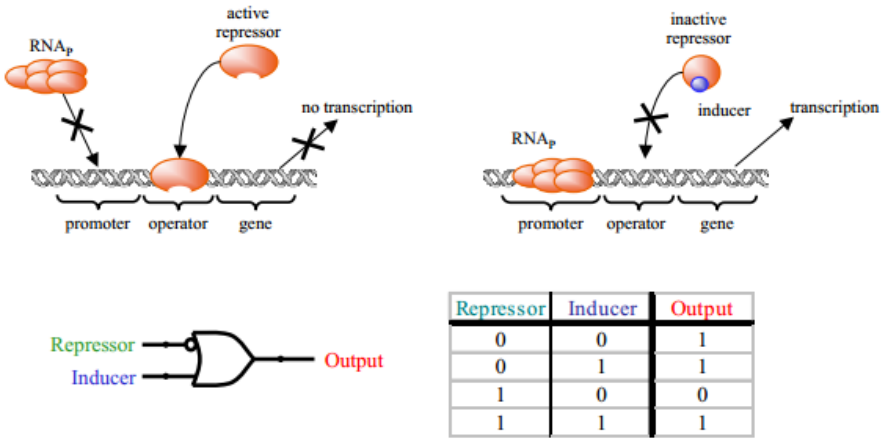


FIGURE 2.8: **Biological NOT gate.** When there are no input proteins present, DNA polymerase binds to the DNA strands and creates the corresponding RNA strand. If there are input proteins, they bind to the DNA strand, so that they do not allow RNA polymerase to move across the DNA strand so it cannot create the corresponding RNA molecule.

The previous example is what in Molecular Biology is known as gene regulatory network, since there are certain genes which regulate the activation and deactivation of other genes by means of their corresponding proteins. All this circuits or programs implemented in bacteria work as gene regulatory networks. The next example is shown below:

[2]Source: Weiss, Ron, et al. "Genetic circuit building blocks for cellular computation, communications, and signal processing." Natural Computing 2.1 (2003): 47-84. Permission obtained from Springer



Repressor	Inducer	Output
0	0	1
0	1	1
1	0	0
1	1	1

FIGURE 2.9: **Biological AND gate.** Sometimes RNA Polymerase needs an activator attached to the DNA strand in order to bind to that DNA strand and create the corresponding RNA molecule. It may happen that the activator protein can only be attached to the DNA strand if it is joined to an inducer. Summarizing, RNA can only bind to the DNA strand and create the corresponding RNA molecule when both activator and inducer molecules are present.

A complex genetic circuit inside a cell may fail due to undesired interactions between natural and synthetic genes of its genome. Furthermore, if the circuit has to be modified, the full bacterial genome should be reprogrammed. Programming bacteria entails several design, simulation and implementation cycles in the laboratory till the desired behaviour of the circuit is achieved. Therefore, modifying all the circuitry of the bacterium would entail spending more time and money on its design, simulation and implementation.

Moreover, if a bacterial population is composed by a big amount of bacteria and each of them contains a program, high-scale parallel computing might be performed.

In the following sections, several methods are proposed for implementing Parallel Computing with a bacterial population and Distributed Computing based on inter-bacterial communication, whereby small circuits may be isolated in different bacteria.

2.4 Parallel Computing with bacterial populations

If each bacterium of a microcolony is considered a single computer, high-scale Parallel Computing might be performed using all the bacteria of that population. Some successful experiments applying this idea has been achieved, like solving the Hamiltonian Path Problem[4]. This example is shown in the pictures below [3]:

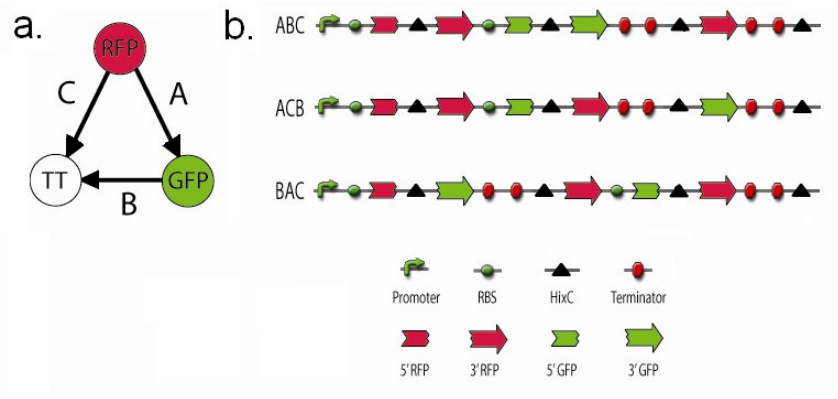


FIGURE 2.10: **Hamiltonian Path Problem consisting on three edged and three possible paths.**

Each of the genetic circuits shown above represent a possible solution to the problem, but only the first circuit is the right one. What does each circuit consist of? There are two genes (RFP and GFP) representing two edges, and two terminators reprsenting the last one (TT). Each of those genes and terminators may be split in two halves so those halves may be autonomously shuffled inside bacteria by a Hin/hixC recombination system, altering the genetic sequence of the circuit. When the proper configuration is achieved after the Hin recombination inside a bacterium (which represents the solution to the HPP problem), it produces a yellow fluorescence as output.

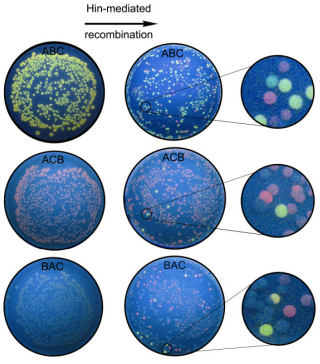


FIGURE 2.11: **Results of the solved problem:** Starting constructions are shown in the left, while the results of Hin recombination are shown in the right.

[3]Source: Baumgardner, Jordan, et al. "Solving a Hamiltonian Path Problem with a bacterial computer." J Biol Eng 3.11 (2009)

2.5 Engineered cell-cell communication

2.5.1 Introduction

In this section, how to design small programs in different bacteria and how to connect those programs to perform Distributed Computing in bacterial populations is explained. Furthermore, if it is desired to alter one of these small programs, it would only be needed to modify its corresponding bacterium instead of the full colony. But bacteria are not connected by wires unlike electrical logic gates, so, how are they able to exchange information? They dispose of two mechanisms:

- Quorum Sensing (QS), which consists on bacteria releasing small molecules into the medium where they are placed for other bacteria to receive those molecules and behave consequently.

- Horizontal Gene Transfer (HGT), whereby bacteria exchange genetic material by means of temporal "wires" between them (sexual reproduction) or using infectious viruses as messengers.

This project consists on HGT using virus as messengers, although both communication mechanisms are described in the following sections in order to explain the advantages of using Horizontal Gene Transfer instead of Quorum Sensing.

2.5.2 Engineered cell-cell communication based on Quorum Sensing

Bacteria have a common way to communicate between them, which consists diffusing molecules to the medium through their membrane, so that other bacteria can receive those molecules and act accordingly to them[5]. Furthermore, receiving bacteria will not take up those molecules till the concentration of such molecules exceeds a certain threshold (Figure 2.12[4]). Gram-positive bacteria use peptides as diffusible molecules, while Gram-negative bacteria use N-Acyl homoserine lactones (AHLs or N-AHLs).

This communication system provides two important advantages:

- **Modularity and Reusability:** As it was explained previously, if it is desired to alter one of these small programs it would only be needed to modify its corresponding bacterium instead of the full colony. Furthermore, those programs may be used in other circuits without spending time and money on their redesign.

[4]Source: Moreno, Angel Goni. "Communication architectures for algorithmic computing in multi-strain bacterial communities." (2010).

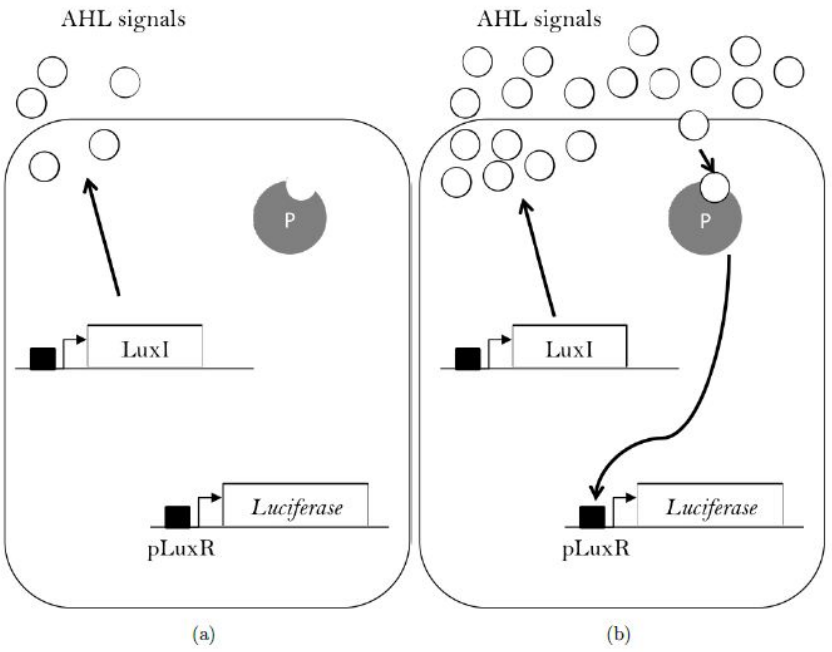


FIGURE 2.12: **Quorum Sensing:** Bacteria release AHL signals for other bacteria to receive them and act consequently. When the concentration of AHL molecules exceeds a certain threshold, such signals go into bacteria so that they induce an activator by binding to it. That activator then binds to the pLuxR promoter, resulting in the expression of Luciferase as output.

- **Robustness:** Cells do not usually behave as expected as they are subject to mutations or modifications in the interactions of their genes and proteins, due to the stochastic nature of living organisms. That is why using two single bacterium (two NOR gates-bacteria, for example) will not always give the expected results. If two bacterial colony (one for each logic gate) made of one hundred bacteria is used instead of single bacterium, it will happen that at least 75 (for example) of those bacteria will release the expected output. Based on it, a threshold may be adjusted in each bacterium so that the second NOR gate-bacteria will receive the molecule from the first NOR gate-bacteria when its concentration exceeds a threshold corresponding to, lets say, at least 60 NOR gate-bacteria releasing such molecule[6].

Another advantage which may be achieved by means of Quorum Sensing is Spatial Computing, whereby a bacterial colony may take several shapes and perform computations following certain spatial patterns. Edge detection is an algorithm widely used in Artificial Intelligence and Image Processing, but it has also been implemented using Quorum Sensing [7]. The bacterial colony is able to release a substance if they do not receive light, and bacteria which sense it will release a fluorescent reporter if they also sense the chemical substance from the bacteria in the dark (AND operation consisting in the light and the chemical substance as inputs and the fluorescent protein as output). This method allow bacteria to form useful patterns to build biomaterials, for example.

The lack of this method is that it only allows to exchange small messages (biological 0's or 1's) depending on the output of the logic gate. What if bacteria exchange bigger messages? Could it be possible to exchange full bioprograms to implement agent-based computing?

2.5.3 Engineered cell-cell communication via DNA messaging

How is it possible that bacteria exchange genetic material i.e. DNA? There are three methods: transformation, conjugation and transduction [8]:

Transformation: There may be DNA fragments in the surrounding medium of bacteria, so that they may take such DNA and incorporate it into their genome. Bacteria which can take this DNA are called bf competent, and those which have already taken those fragments are called transformants(figure 2.13 [5]).

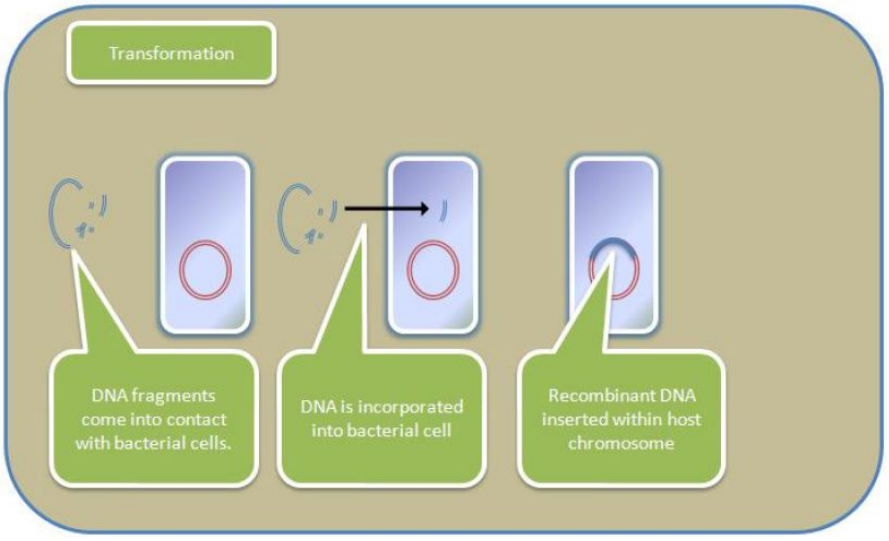


FIGURE 2.13: Transformation

Conjugation: Bacteria use a reproduction mechanism which consists on the exchange of double-stranded, circular DNA molecules called plasmids(figures 2.14 and 2.15 [6]) through a temporal "wire" known as mating pair or pilus. These molecules encode the necessary machinery for its exchange between bacteria and they also encode other genes which may be included by means of Genetic Engineering techniques.

Transduction: This communication mechanism is based on the exchange of nucleic acids (DNA or RNA) by means of a type of viruses called bacteriophages. Those viruses

[5]Source: Prestes Garcia, Antonio. A first approach to individual-based modelling of the bacterial conjugation dynamics. Diss. Informatica, 2011.

[6]Source: Prestes Garcia, Antonio. A first approach to individual-based modelling of the bacterial conjugation dynamics. Diss. Informatica, 2011.

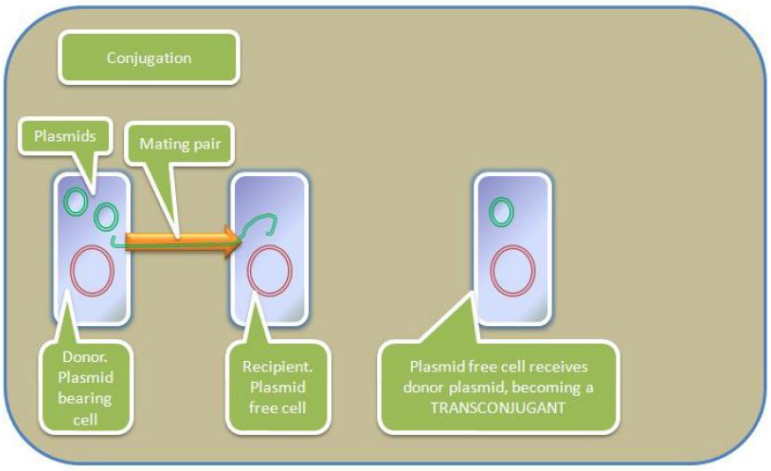


FIGURE 2.14: Conjugation

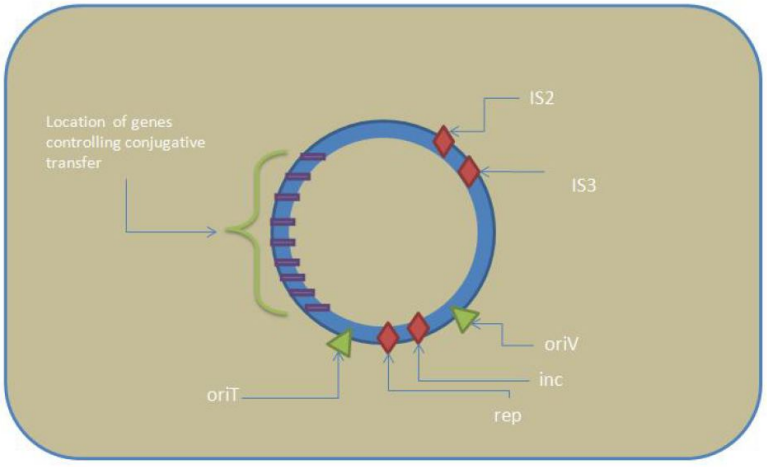


FIGURE 2.15: **Plasmid schematic view with relative gene positions regulation the plasmid functions:** IS2 and IS3 are the insertion sequences that regulate the plasmid insertion into the host chromosome. The terms inc and rep are the genes responsible for plasmid replication and incompatibility, oriV and oriT are the origin o replication and the origin of transfer respectively. In the left side is the operon which regulates transfer functions.

insert their genetic material into bacteria, so that it replicates itself using the cellular machinery and new viruses are released from such bacteria. This process may be carried out with lysis (the breaking down of cells) or without lysis. From a Computer Engineering point of view, this communication mechanism may be seen as the actual wireless systems of conventional computers.

Transduction with lysis: There are some bacteriophages, such as the lambda phage (figure 2.16[7]), which replicate themselves into cells till a certain amount of copies is

[7]Source: Labrie, Simon J., Julie E. Samson, and Sylvain Moineau. "Bacteriophage resistance mechanisms." Nature Reviews Microbiology 8.5 (2010): 317-327. Permission obtained from Nature Publishing Group

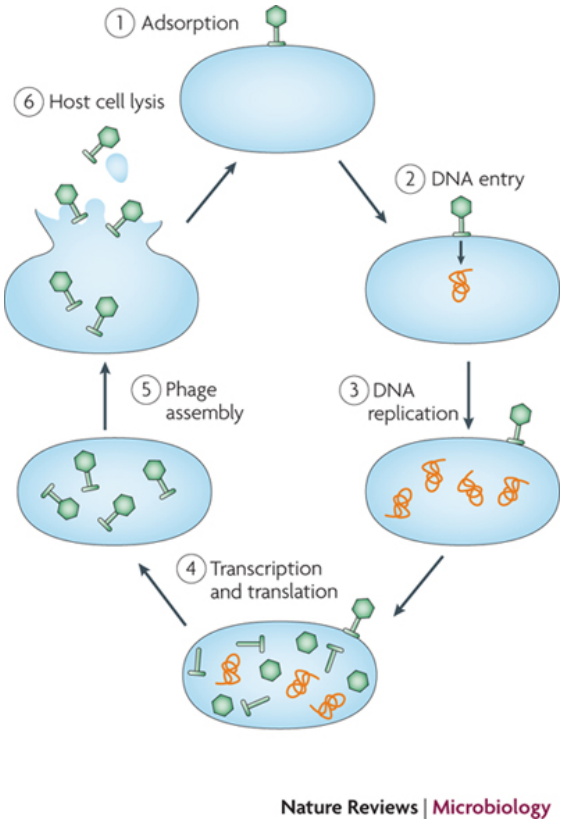


FIGURE 2.16: **Transduction with lysis**

reached. Then, the cellular membrane can not withstand the pressure exerted by those copies so that it bursts, releasing all the new viruses.

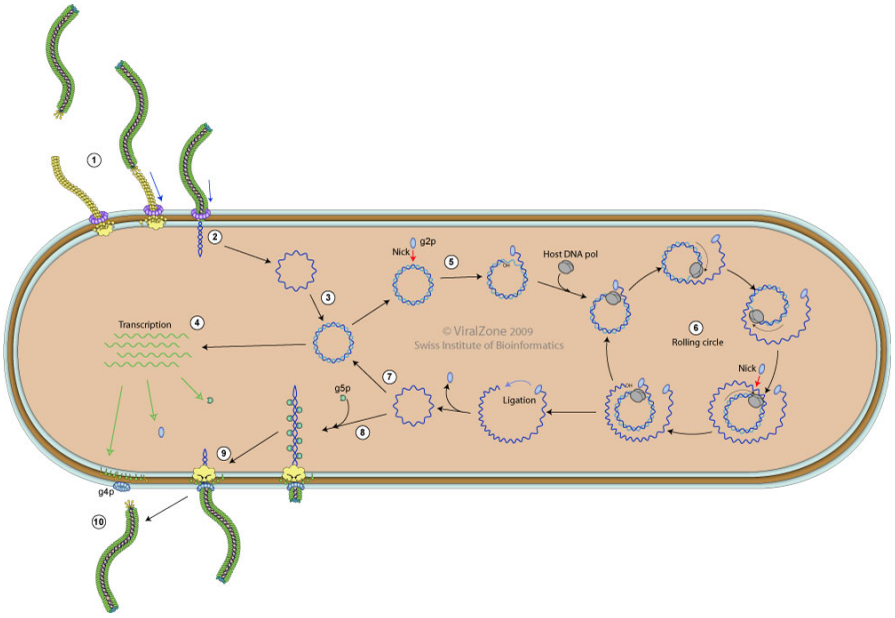


FIGURE 2.17: **Transduction without lysis**

Transduction without lysis: There are, however, other bacteriophages which do not burst the cellular membrane, since they are continuously released after a certain amount of time (figure 2.17[8]). Such viruses are called filamentous bacteriophages (they are described in the following section), and that is one of the main reasons why they are used in this project.

2.6 Filamentous bacteriophages

In this section, the F-specific filamentous phage (Ff) is described. It is named in that way because it depends of the F-pilus (which bacteria use to exchange genetic material via conjugation) for infection.

There are three main types of Ff phages which are 98% homologous: f1, fd and M13 [9]. The last one is the Ff phage used in this project so it is described in the following subsections.

2.6.1 Structure of the M13 bacteriophage

This virus consists of a circular single-stranded DNA molecule encapsulated in a two protein coats: minor and major.

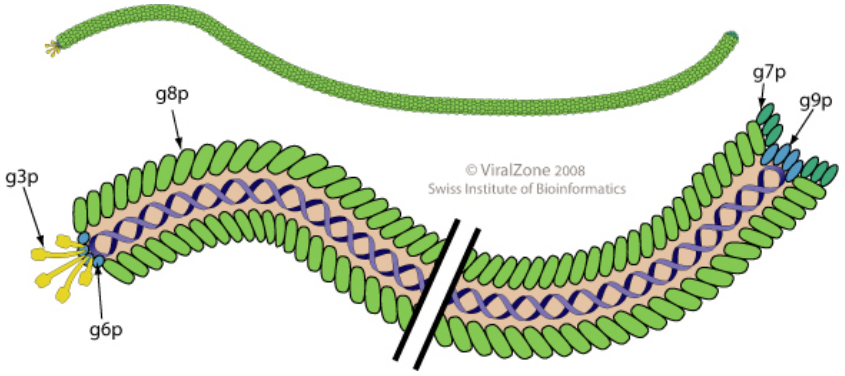


FIGURE 2.18: **Structure of the M13 bacteriophage**

The **minor coat** is in both right and left ends of the phage (proximal and distal end, respectively), where proteins p9 and p7 are placed in the right end, while p3 and p6 are in the left one (figure 2.18 [9]).

The **major coat** protein consists of 2700 copies of the protein p8.

[8]Source: ViralZone www.expasy.org/viralzone, Swiss Institute of Bioinformatics
[9]Source: ViralZone www.expasy.org/viralzone, Swiss Institute of Bioinformatics

2.6.2 Infection mechanism with M13 in Escherichia Coli bacterium

Bacteria have several pilus which they use for the sexual reproduction. Such pilus are randomly retracted, so that bacteriophages adhered to the pilus by means of the protein pIII can reach the bacterial surface. When it happens, viruses insert their single-stranded DNA (positive strand) into bacteria and their major coat proteins p8 enters the cytoplasm. RNA polymerase binds to such DNA strand and creates a RNA primer which DNA polymerase III uses to replicate the complete negative strand, resulting in a circular, double-stranded viral DNA called replicative form (RF). Protein pII then binds to this dsDNA molecule and starts creating replicas of the positive strand, one at a time. Those positive strands can be replicated to form new dsDNA RF, which serve as a template to create phage proteins needed to assembly viruses, or they may be directly packaged into phage proteins to be released as new bacteriophages. Figure 2.17 explains this mechanism.

There is an important element in the assembly of new viruses into bacteria: the packaging signal. This DNA segment is located in the RF, and it is the only exposed segment of the genome because it is a hairpin loop. Such packaging signal has to interact with the assembly machinery (proteins p7, p9 and p1) to initialize this process, so if it is replaced with a hairpin loop of different sequence, the assembly is prevented. This may be dangerous for bacteria, because the over-expression of certain genes in the absence of the assembly may cause cell death.

Some quantitative data obtained from experiments state that bacteriophages are assembled into bacteria, but they are not released until 15-20 minutes. Then, the production of viruses is exponential for the first 60 minutes, after which it becomes linear (bacteria are in stationary phase). It has been experimentally verified that 1000 phage/bacterium are released for the first hour.

Other important point are the alterations produced in an infected bacterium:

- Its energy consumption increases, so they grow and divide slower than uninfected cells.

- Acid Resistance genes for low pH conditions were down-regulated on protein level, so these bacteria are less tolerant to low pH, which consequently makes infected cells more sensitive to harmful conditions.

- The phage protein p4 is also inserted into the cell membrane when a virus infects a bacterium. This protein gives rise t the expression of an Escherichia Coli phage shock protein (PspA).

It is also important to know that those alterations are produced just the first time bacteria are infected by a first bacteriophage. This means that consecutive infections will not alter cells any more.

2.6.3 Formal representation of the infection process

The previous subsection described the infection mechanism of the M13 bacteriophage in Escherichia Coli bacteria giving importance to low-level biological processes involved on it, but it may be abstracted in a simpler model which can be implemented in-silico to simulate this mechanism (figure 2.19 [10]). Such abstraction is the main topic of this subsection.

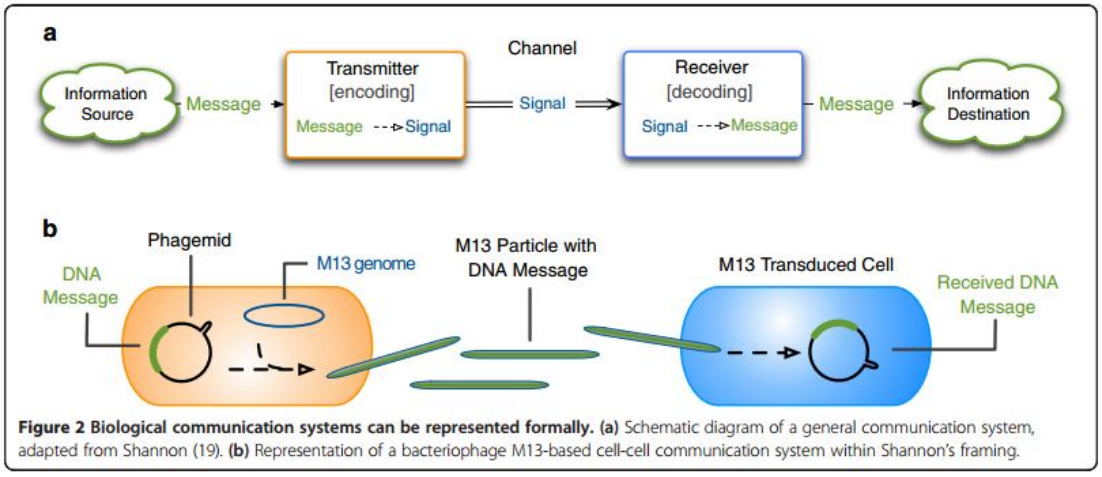


FIGURE 2.19: **Biological communication systems can be represented formally**

All the previously explained phases of the infection mechanism can be abstracted in three steps (figure 2.21 [11]): (1) Phage inoculation. (2) Latent phase. (3) Release of phages:

Inoculation: Initial instant in which phages start searching for cells which they can bind to and infect.

Latent phase:

- Search time (ts): The time needed for a phage to infect a cell. It requires them to find cells which they can bind to.

-Infection: The most important fact in this phase is the probability of infection. Bacteria can be infected by phages surrounding them in every time step. The radius in which

[10]Source: Ortiz, Monica E., and Drew Endy. "Engineered cell-cell communication via DNA messaging." Journal of biological engineering 6.1 (2012): 1-12.

[11]Source: Charlotte Maestracci. "Simulation of synthetic ecosystems of bacteria and phages". Internship report (2012)

they are infected is one um plus its size, which means that every phage placed in its eight neighbour patches plus its own patch can infect the bacteria in every time step (figure 2.20 [12]).

There are several operations which have to be taken in consideration when calculating the probability of infection, which is considered the same in both infected and uninfected cells:

Number of virions that cells receive: This number follows a Poisson distribution:

$$P(k) = \frac{e^{-m} * m^k}{k!}$$

Where: m is the Multiplicity of Infection (MOI), i.e., the ratio of phages that infect bacteria after the inoculation phase with a period T. k is the number of phages infecting one bacterium P(k) is the proportion of cells that will receive k phages

Period T is assumed to be equal to the search time (ts), because it corresponds to the time phages are looking for bacteria to infect them.

Proportion of infected cells by at least one phage after inoculation during ts:

$$P_{inf} = P(K >= 1) = 1 - P(K = 0) = 1 - \frac{e^{-m} * m^0}{0!} = 1 - e^{-m}$$

Proportion of infected cells by at least one phage after inoculation during one minute, following the Poisson law, is:

$$P_{inf} = P(K >= 1) = 1 - P(K = 0) = 1 - e^{-m/ts}$$

This global behaviour can be adapted to local behaviour using the local MOI instead of the global MOI, so:

Local Multiplicity of Infection (Local MOI): It is the ratio between the phages surrounding the cell, i.e., in radius one, to the amount of cells which those phages can infect. Those cells are in radius 1 of these phages, so they are in radius 2 of the studied cell:

$$LocalMOI(ts) = ml_{ts} = \frac{\text{phages in radius 1 from the studied cell}}{\text{cells in radius 2 from the studied cell}}$$

Using the local MOI, the proportion of cells that will be infected by these pahges during one time step is:

$$P_{inf-local} = P(K >= 1) = 1 - P(K = 0) = 1 - e^{-ml/ts}$$

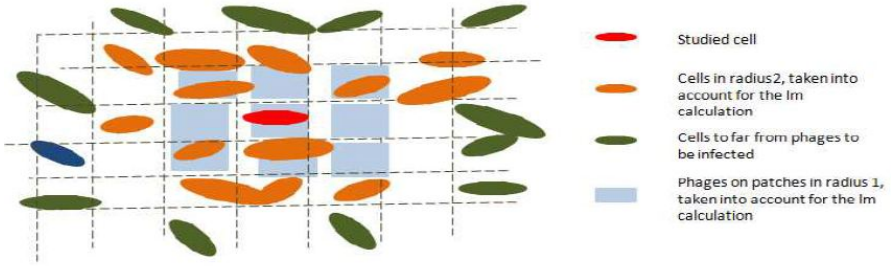


FIGURE 2.20: **Cells and patches involved in the calculation of the Local MOI**

This value can be used to calculate the number of phages that infect the cell by generating a random number x between 0 and 1. If $x <= P$, the cell is infected, which means that the number of phages that infect it is equal to the local MOI *ml*. The cell will absorb ml phages randomly from the surrounding patches of the cell, so they will disappear.

- Eclipse time: This is the phase in which enough virions are synthesized into the cell to be released.

Release of phages: This function is controlled by the infection clock. When a cell is infected, its infection clock is set to 0 and it increases in every time step till its value is bigger than the eclipse time. Then, the cell starts releasing phages randomly between its neighbouring patches with a rate of 100 phages per hour and per cell.

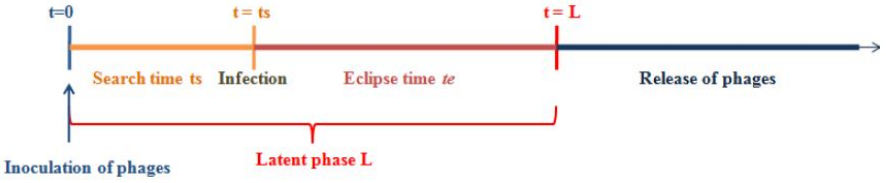


FIGURE 2.21: **Representation of the infection as sequential phases**

Changes in infected cells: The first time a cell becomes infected it changes the length of its cell cycle multiplying it by the cycle length ratio ra:

$$ra = \frac{cycle - lengthofinfectedcells}{cycle - lengthofuninfectedcells}$$

which depends on experimental conditions and has been set to 1.25 as default value.

Phages diffusion in agar gel: This value depends on the concentration of Agarose and it is represented as a coefficient D which is defined by the user. Therefore, the movement of phages is dictated by the following Brownian law:

$$< X^2(t) >= 2 * dim * D * t, \text{ so that}$$

[12]Source: Charlotte Maestracci. "Simulation of synthetic ecosystems of bacteria and phages". Internship report (2012)

$\sqrt{< X^2(t) >}$ is the quadratic moving average dim is the dimension of space D is the diffusion coefficient t is the passed time

Using the previous formula it can be concluded that:

$$d = \sqrt{2 * 2 * D * 60} * 10^6 = 2 * 10^6 * \sqrt{60 * D}$$

where d is the distance covered by each phage during one time step.

This means that each phage is randomly distributed among the neighbour patches in a distance d from its original position.

2.6.4 Phage display technology

This application of phages consists on the inclusion of genes into the phages genome to allow them synthesizing proteins which can be displayed in their coat proteins. This allows studying proteins by doing the following procedure:

1. The corresponding genes of the protein which is being studied are inserted into the phage genome, so that they display such protein in their coat.

2. When phages are released from host cells in sufficient numbers, there will be a certain amount of protein variants due to mutations. Such variants may have a different binding targets or increased affinity to targets of interest.

3. Phages displaying a stronger protein will bind to a solid support containing the ligand of interest immobilized in its surface.

4. Washing is then applied to eliminate phages which display non-binders.

The previous steps of binding and washing are performed at least three times because there are some non-specific binders which still remain in the surface, so that in the end the stronger binder is the only kind of protein that is still bound to the surface.

All the coat proteins of the M13 bacteriophage have been used as platform for phage display, although the minor protein pIII (which determines the infectivity of the phage) is the most commonly used, so does this project in order to process information.

2.6.5 Phages and phagemid vectors

There are two ways to display proteins using phages, so they are showed here using pIII display[10]. The first one consists on using the complete phage vector genome inserting

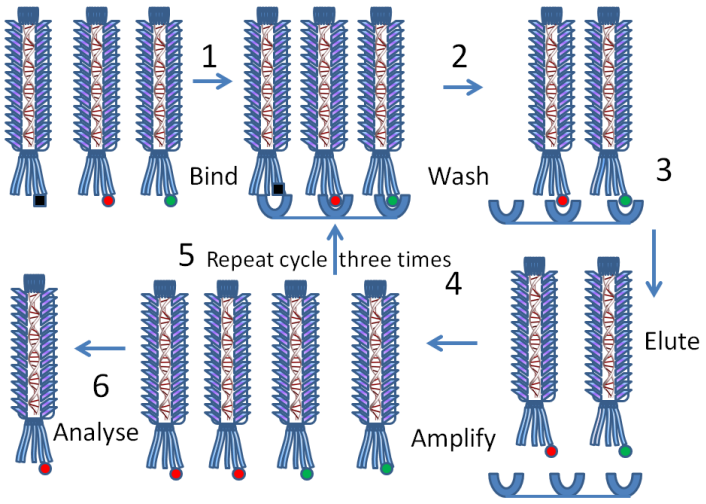


FIGURE 2.22: **Phage display process**

the sequence of the displayed protein into the coding region of the coat protein pIII. When these phage vectors are inserted in E.Coli, phages containing all the PIII coat proteins fused to the desired protein are produced.

The second way consists on using a small plasmid-based vector called phagemid. These plasmids are under the control of a weak promoter and they contain:

- A plasmid origin of replication (oriV), which correspond to a particular case called pBR322 ori.

- An Ff origin to allow the production of single stranded vectors (synthesized from this plasmid) and their packaging into phage particles.

- The fusion protein, i.e., the displayed protein and the pIII protein sequences.

As the phagemid only contains the sequence of the fusion protein, the other phage particles need some source in order to be synthesized. Such source is a helper Ff phage introduced into bacteria, so they can produce all the needed phage proteins for the production of new virions. This helper phage has a reduced packaging efficiency because their packaging signal is disabled[11], ensuring that phagemids are preferentially packaged. Note that the resulting phages contain a big amount of the wild type pIII protein as they are also synthesized from the helper phage like the other phage coat proteins.

2.7 Modelling bacterial populations

2.7.1 Individual-Based Models vs Equation-Based Models

One of the first questions that should be asked before simulating any complex system is what model fits better with the system to be simulated. There are two main types of models widely used to simulate bacterial populations: Equation-Based Models and Individual-Based Models (IBM, also known as Agent-Based Models or ABM)[12].

Equation-Based Models are widely implemented with differential equations. They capture the characteristics of the simulated system by identifying its variables and describing their behaviour with a set of equations, so that the execution of this kind of models consists of solving them.

One of the common and simplest examples to illustrate this idea are Lotka-Volterra predator-prey equations, which describe the interaction between two populations that show predator-prey behaviours:

$$\frac{dA}{dt} = A(r - \alpha B)$$

$$\frac{dB}{dt} = B(-\sigma + \beta A)$$

Where:

- t represents time,

- A(t) represents the size of the prey population at time t,

- B(t) represents the size of the predator population at time t

- α,σ and β are parameters describing the interaction between the two species.

The results of solving these equations are the behaviour of variables A and B over time, which can be viewed by means of graphics:

On the other hand, Individual-Based Models consist of a set of agents defining several parameters, a set of global parameters and a set of rules which are applied to each agent's parameters and global parameters in each time step to simulate interactions between all the individuals included in the model. Those interactions give rise to an emerging behaviour of the population of individuals, which is the counterpart of the simulation results of differential equations.

Building a Predator-Prey model with IBM would involve the following steps:

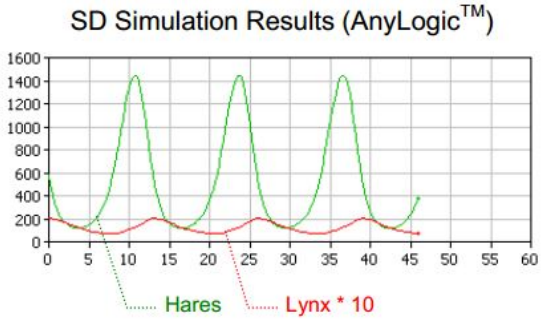


FIGURE 2.23: **Simulation results of the predator-prey model using differential equations**

1. Identify the rules that determine the behaviour of populations A and B:

Agent	Rules
Hare	• Move randomly in the world. • Check to see if you should reproduce this time. • Check to see if you've reached maximum age.
Lynx	• Move randomly in the world. • Check to see if you landed on a hare to eat. • Check to see if you should reproduce this time (do you have enough energy to have a kitten). • Check to see if you've reached maximum age. • Check to see if you're out of energy.

FIGURE 2.24: **Outlining the behaviour of Hares and Lynx**

2. Identify parameter values of each population based upon the rules:

Parameter Name	Minimum Value	Maximum Value	Initial Value
initial-hares	0	1000	100
initial-lynx	0	100	20
hare-birth-rate	0	200	35
max-hare-age	0	20	6
lynx-energy-to-reproduce	0	100	30
energy-per-hare-eaten	0	30	10
max-lynx-age	0	50	20

FIGURE 2.25: **Possible parameter settings**

3. Develop an algorithm to evaluate each individual of the model. Such algorithm evaluates the parameters of each individual and also the global ones, modifying each individual in function of these parameters. This algorithm uses the defined rules in the step 1.

The following graphic shows the simulation results of this Individual-Based Model of predator prey populations, which fits better with reality:

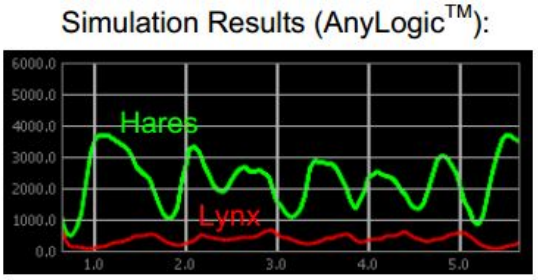


FIGURE 2.26: **Simulation results of the predator-prey model using individual-based models**

Why are Individual-Based Models widely used? Equation-Based models show limitations when capturing individual properties, which cannot be fully taken into account in state variable models. Individual-Based Models not only allow capturing those individual properties, but also allow understanding how individual properties determine the overall system properties. Furthermore, Individual-Based Models can be improved by allowing agents to have the capability of learning, which means they can adapt to the environment. Such improvement can be achieved by implementing neural networks or evolutionary computation techniques, like genetic algorithms or genetic programming.

The software tool improved in this project uses Individual-Based Modelling to simulate bacterial populations, considering both bacteria and bacteriophages as agents of the system.

Equation-Based Models can be represented by means of mathematical formulas, but Individual-Based Models are more descriptive. How can they be represented? There are no established standards, but there is a widely used protocol to do so, which is presented in the following subsection.

2.7.2 ODD protocol

Individual-Based Models were difficult to understand and evaluate as they were poorly documented, so it was needed some kind of standardization to make them more understandable and easier to reproduce [13]. That is the reason why the Overview, Design Concepts and Details (ODD) protocol was created and published. This protocol makes model descriptions more structured, complete and efficient, so that Individual-Based Models became more commonly used.

It consists of the following seven elements which are categorized in Overview, Design concepts and Details:

2.7.2.1 Overview

Purpose: This first section does not describe how the model is implemented, but the summary of objectives for which it has been developed.

Entities, state variables and scales: This section explains what kind of entities are in the model and what state variables and attributes characterize these entities. Some examples of entities and their attributes are classified as follows:

Entity	**Example**	**Possible attributes**
Agents	Bacteria, viruses	Coordinates, length, cell type
Spatial units	Grid cells	soil moisture content, soil nutrient concentration
Environment	Temperature	Maximum and minimum temperature
Collectives	Bacterial strand A, bacterial strand B	Maximum and minimum number of agents

TABLE 2.1: Examples of entities, state variables and scales

Process overview and scheduling: This section contains the pseudo-code which specifies what the actions performed by the entities are, in what order they are performed and how time is modelled, for example.

2.7.2.2 Design concepts

Basic principles: General concepts, theories, hypotheses, and modelling approaches underlying the design of the model.

Emergence: Model results expected to vary when characteristics of individuals or their environment change.

Adaptation: Rules that individuals use to make decisions in response to changes in their environment or themselves.

Objectives: Organisms can adapt to achieve some fitness or objectives, which should be listed here.

Learning: How individuals change or change or adapt over time.

Prediction: Learning procedures might be based on estimations about future conditions. This section should describe how individuals perform those predictions.

Sensing: Internal and environmental state variables that individuals take into consideration when making decisions.

Interaction: How individual influence each other by means of physical contact, for example.

Stochasticity: What processes are assumed to be random and how their randomness is implemented.

Collectives: When individuals form groups, they can be a new kind of entity with its own state variables, but they can also be emergent properties coming from the individual behaviours.

Observation: This section indicates what data are used for analysing the models.

2.7.2.3 Details

Initialization: This section defines the initial state of a simulation run, specifying if it always takes the same value or if it can vary between several simulation runs.

Input data: The model can use external sources like data files or other models to take inputs, so such sources should be specified here.

Submodels: The processes listed in 'Process overview and scheduling' are composed of submodels which should be described here, including their parameters, values and how they were tested. This section might also include model fitting and experimental validation.

2.7.3 Parallel Computing for massive populations

Simulating a bacterial population modelled by means of Individual-Based Models entails that each of the individuals has to be evaluated and modified in each time step. The simulation is possible by using conventional tools if the population is small, but realistic simulations have to deal with big amounts of individuals, representing a large computational cost. When this project is achieved, it will be concluded if parallel computing is needed to simulate big amounts of bacteriophages.

This practise is widely used in several disciplines of bioinformatics, neuroscience or biomedicine because biological systems are complex even for a computer. The Human Brain Project (http://www.humanbrainproject.eu/) aims to simulate all the brain regions, which are composed of large amounts of neurons. Taking into account that each

neuron is complex, simulating the entire brain will require not only a supercomputer, but several supercomputers sharing and processing information during simulations.

2.7.4 Scope of the simulations

This subsection summarizes the main characteristics which may be taken into account when simulating bacterial populations. Such characteristics and variables can be divided in: (1) internal state of bacteria, (2) state of the physical media including its substances, (3) dynamics of physical interaction between bacteria and (4) inter-bacterial communication. A table showing these characteristics is shown below:

Internal state of bacteria	Shape
	Elongation process
	Cell Division
	Grow speed
	Cell death
	Intracellular process (including the state an quantity of its genes and proteins)
State of the physical media including its substances	Pressure
	Viscous drag
	Biofilms
	EPS
	Noise
	Shape of the physical media
	Size of the physical media
	Chemostat mode
Dynamics of physical interaction between bacteria	Shoving caused by the movement and pressure exercised between bacteria
	Shoving caused by a bacterium which divides itself because of the cellular division cycle
Inter-bacterial communication	Quorum Sensing
	Bacterial Conjugation
	Bacteriophages

2.8 Simulation tools

2.8.1 Introduction

Bacterial simulations have been very useful in that they safe a lot of money and time when researchers want to perform a new experiment. They are not only important in Synthetic Biology, but they are essential because synthesizing new bacteria is expensive and not all researchers dispose of a wet lab to perform experiments. There are several biological processes and phenomena interesting for engineering organisms, like bacterial conjugation, bacterial biofilms, intercellular interactions or intracellular genetic networks. Bacterial Computing is a subfield of Synthetic Biology which tries to implement computing models in biological substrates using the processes named above, and bacterial simulations are needed to check if those models work properly.

Population-Level Models (PLM) like differential ecuations used to be applied to simulate bacterial colonies, but they were not accurate enough because they only focused on the population variables without taking in account individual behaviours. Simulations were improved when Individual-Based Models[14] (IBM, also know as Agent-Based Models or ABM) emerged. Those models take into account the behaviour of each agent in the system modifying the values of its variables. Then the population-level emerges from those individual behaviours, turning bacterial simulations more accurate.

There are several simulation tools for this purpose, like DiSCUS, GRO, CellModeller, BSim and iDynoMICS. All of them are developed for similar purposes, but there are some differences in the following features:

- Scope of the simulations
- What models they are based on
- What levels of abstraction they provide
- Technical features
- How easy to extend they are
- Computational performance

Those features are explained for each of the simulators in the following sections. There is also a summary which gathers this comparison at the end of this chapter.

2.8.2 DiSCUS

This simulation tool has been developed by Angel Goni and Martyn Amos [15] using the Python language. It is essentially designed to simulate a population of bacteria which interact between them by plasmid conjugation and shovings. Physical interactions are measured taking into account several parameters and phenomena like the shape of the cells, their radius, the cellular division and the pressure exerted between them. This leads to the definition of new types of cells being simulated. Conjugation processes are based on the time they need to be performed, and they also affect the biophysics because of the influence made by the elastic springs which keep two bacteria joined.

It can also simulate intracellular genetic networks in a modular fashion, allowing to implement the desired circuit regardless the intercellular parameters of the simulation. In addition, the shape of the 2D medium in which bacteria are interacting can be easily modified, because it is defined by a Python function that can be written by the user (figure 2.27).

```
def add_walls(space):
### Function used to place the walls in the world (walls scaled depending on screenview)
### In: the space
### Out: returns the shapes (l) of the walls
    body = pymunk.Body()
    body.static = True
    body.position = (0.39*screenview,0.39*screenview)
    l1 = pymunk.Segment(body, (-0.39*screenview, 0), (0.0, 0.0), 5.0)
    l2 = pymunk.Segment(body, (0.0, 0), (0.0, -0.39*screenview), 5.0)
    body = pymunk.Body()
    body.static = True
    body.position = (screenview - 0.39*screenview,0.39*screenview)
    l3 = pymunk.Segment(body, (0, 0), (0.39*screenview, 0.0), 5.0)
    l4 = pymunk.Segment(body, (0.0, 0), (0.0, -0.39*screenview), 5.0)
    body = pymunk.Body()
    body.static = True
    body.position = (0.39*screenview,screenview - 0.39*screenview)
    l5 = pymunk.Segment(body, (-0.39*screenview, 0), (0.0, 0.0), 5.0)
    l6 = pymunk.Segment(body, (0.0, 0), (0.0, 0.39*screenview), 5.0)
    body = pymunk.Body()
    body.static = True
    body.position = (screenview - 0.39*screenview,screenview - 0.39*screenview)
    l7 = pymunk.Segment(body, (0, 0), (0.0, 0.39*screenview), 5.0)
    l8 = pymunk.Segment(body, (0.0, 0), (0.39*screenview, 0.0), 5.0)
    space.add(l1, l2, l3, l4, l5, l6, l7, l8)
    return l1,l2,l3,l4,l5,l6,l7,l8
```

FIGURE 2.27: Funtion defining the shape and size of the physical media

Agent-Based models are implemented in this software to simulate intercellular interactions between bacteria. Each individual has a behaviour based on several attributes, like shape, elongation, position, speed and so on. For the time being, only ordinary differential equations written in Python have been used to simulate intracellular processes, but this tool allows to implement any desired program into the cell. This leads to high abstraction of intracellular processes. For example, an OR gate can be implemented with the differential equations which describe it, but it can also be implemented just

with a simple OR operation using the logical operators of Python. There are also several global parameters, like width, length and growth speed of the cells, the number of steps of the ODEs per doubling time of the simulated cells, or friction coefficient, among others. They are used to define the features of the simulation.

This simulation tool can be executed in Windows, Linux or Macintosh because it is written in Python, a powerful and multiplatform programming language which is not very difficult to learn for users who have not programmed before. It needs several packages to be executed, which are Pygame, Scypy, NumPy and Pymunk. Those packages are actually made for windows 32-bit version, so a version of Python for windows 32-bit must be used, although a windows 64-bit is being used.

Its operation is based on Python line-commands to run it, and the parameters needed to establish the features of the simulation can be modified by Python code. Its outputs use the Pygame library to show graphycally how bacteria interact and grow (figure 2.28).

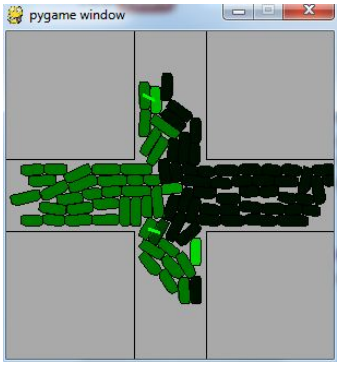


FIGURE 2.28: Pygame graphical interface

As it is developed with Python, it allows to extend the tool with new functionalities, like the simulation of new biological processes or the inclusion of a User Graphical Interface to establish the simulation parameters in a more easy and convenient manner for non-programmers.

The main algorithm follows some steps in each iteration of its main loop. Each iteration is applied to each individual in the model, so this can get very computationally expensive for a big number of individuals. Intracellular processes has only been proved with deterministic ODEs, but using stochastic ones could worsen the performance. Parallelizing intercellular processes and implementing parallel algorithms to solve the future stochastic equations that can be added could be a solution.

2.8.3 GRO

This programming language has been developed by Seunghee S. Jang, Kevin T. Oishi, Robert G. Egbert, and Eric Klavins [16] in order to extend classical bacterial simulators, which only focused in single-cell simulations of gene regulatory networks without taking into account physical interactions between cells.

GRO's simulations covers the bacterial growth, molecular signalling by quorum sensing, physical forces like shovings, noise and a physical media in chemostat mode. Agents are not restricted to be bacteria, because this language uses IBMs to model cells, so they are composed of several attributes which can be modified in order to describe different kinds of organisms. GRO can also model edge detection and programmed morphology in bacterial populations due to its capability of simulating molecular signaling.

The main feature of this language is that it is based on guarded commands (figure 2.29). They consist on sentences of the form g:c, where g is a Boolean expresion and c is a list os statements. All intracellular processes, like production or degradation of chemicals, cell growth or cell division can be implemented with this commands. For example, noise can be simulated by using a function rate() in the guard command which is supposed to be stochastic. They also allow to run the programs using paralellism, which means they are more efficient.

```
include gro

set ( "dt", 0.01 );

alpha_r := 69.4 / 2.35;          // mRNA / min / fL
beta_r := - log ( 0.5 ) / 3.69; // 1/min
alpha_p := 3.0;                  // protein/min/fL/RNA
beta_p := 0.01;                  // 1/min

program gfp() := {

  mRNA := 0;
  gfp := 0;

  rate ( alpha_r * volume ) : { mRNA := mRNA + 1 };
  rate ( beta_r * mRNA ) :    { mRNA := mRNA - 1 };
  rate ( alpha_p * mRNA ) :   { gfp := gfp + 1 };
  rate ( beta_p * gfp ) :     { gfp := gfp - 1 };

};

set ( "gfp_saturation_max", 1000 );
set ( "gfp_saturation_min", 800 );

ecoli ( [ x := 0, y := 0 ], program gfp() );
```

FIGURE 2.29: GRO code

The physical media consists of a 2D grid whose size and shape can be modified by code, in a similar manner to DiSCUS.

GRO also allows to implement biological processes with the desired level of abstraction, due to the flexibility of guarded commands, which can perform a differential ecuation or

a simple Boolean expresion. This idea comes from a top-down approach, which states that a biological design can be developed thinking about high-level behaviours of the system, and then go down to develop more basic functionalities.

This language is implemented with lex, yacc and C++. It uses the chipmunk 2D physics library for the physical interactions and Qt for the graphics (figure 2.30).

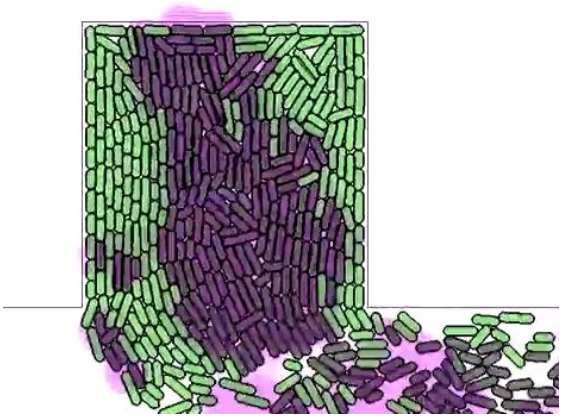


FIGURE 2.30: GRO graphics

Simulations performed by this language can get very computationally expensive when it has to simulate a very big number of individuals, because its main algorithm is not as parallelized as needed (only intracellular processes are parallel because of the guard commands). As was proposed for DiSCUS, improving the parallelism is the solution for this language.

2.8.4 CellModeller

This is another tool based on Python. It has been developed by Timothy J. Rudge, Paul J. Steiner, Andrew Phillips and Jim Haseloff [17] in order to simulate intracellular processes, molecular signalling, bacterial growth, physical interactions, bulk flow and advection in a 3D surface.

Cell biophysics are described using IBMs with several attributes which describe the features of the cells. This tool can simulate more complex physical phenomena, like viscous drag which dominates inertia.

In regard to molecular signalling, it can be implemented with differential equations or rules. The last ones could be used to study the effects of growth rate on colony morphology, for example. Thanks to the possibility of simulating this kind of process, edge detection and programmed morphology can be addressed with this tool.

Intracellular process can be simulated with ordinary differential equations, although this tool allows to use stochastic methods running in parallel, which could be a future implementation.

This two last phenomena, molecular signalling and transcriptional regulation are simulated separately, allowing to simulate very modular systems. The only problem of this software is that it can not simulate extracellular polymeric substance, which is common in bacterial biofilms.

This software is based on IBM and it is implemented with Python modules, which allows using the desired level of abstraction. The libraries needed for its use are pyopencl, numpy and scypy, because it is developed to perform parallel computations using both CPU and GPU architectures. Graphics are shown with Matplotlib library.

There is also an idea about developing some module which can read SBML models to use them as intracellular programs, instead of using only ordinary differential equations or simple rules.

This tool uses OpenGL to run simulations in parallel with both CPU and GPU architectures, which are very powerful for this kind of massive simulations. It has been proved that a colony of 32,200 cells was simulated in 30 mins, so it is a very good tool to simulate big amounts of individuals.

2.8.5 BactoSim I

This simulation tool is based on the Java programming language. It is also based on Repast Symphony 2.0, an agent-based modelling system for Java which allows implementing IBMs with flexibility, making the simulator more difficult to develop and modify.

The GUI can be easily modified both by means of XML files or using the options given in such GUI.

IBMs are the models using in this simulation tool. Individuals have several attributes which can be modified in order to create new agents. There an advantage by using Java, because it allows to create subclasses from a parent class.

Regarding to the abstraction levels, Java allow to everyone to develop models as much accurate you want to perform a simulation.

Using it is not easy for non-programmers, because Java is an object oriented language, and it can take a long time to understand the main concepts.

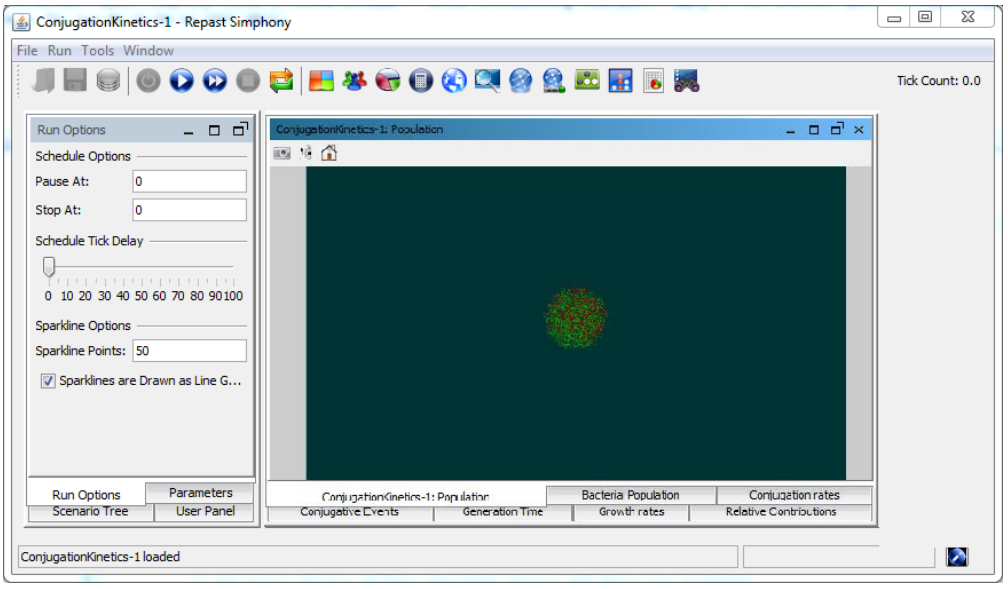


FIGURE 2.31: BactoSim I graphics

The simulation results are plotted in a understandable GUI and stored in .dat files if desired.

This software is implemented with Java and it has not any parallelism, so it does not allow to simulate big amounts of individuals.

2.8.6 iDynoMICS

With this tool[18] more complex biological processes can be simulated. Molecular signalling, physical interactions and intracellular processes are handled in a similar manner as in the previous tools, but there are some extra features. Noise is implemented in a lot of processes, like agent location and mass, cell division threshold or daughter cells size. In intracellular processes, biochemical reactions are decoupled from species, so individuals of a species can perform different sets of reactions, e.g., agents can adapt their metabolic reactions to environmental conditions.

But the main important feature of this simulator is that it is the only which can simulate the presence of extracellular polymeric substance (EPS). It also simulates cell compartments, like active biomass, inert biomass and capsular EPS. The intercellular EPS is simulated as a different kind of agent, so it influences the biophysics, like pressure and shovings. Cells excrete EPS in each agent's step, depositing it in matching EPS agents. If no EPS agents are found, a new one is created.

There also some global parameters which can be selected by the user, like biofilm erosion, biofilm thickness or advection. Some other parameters can not be chosen, like surface

shape, because it only allow to select its size, although is can be a 3D surface which means an improvement about physical media simulations. This physical media has four distinguished regions: Support (inert), Biofilm and boundary layers (both diffusive) and Bulk (convective). This allows to simulate the physical media in a more realistic way. This let to perform simulations in a stochastic chemostat mode.

It is based on IBMs which uses several attributes to model individuals. Those attributes can be modified in order to model new kind of individuals, like different organisms.

Simulations can use files as inputs to initialize variables, and results can be viewed graphically (figure 2.32).

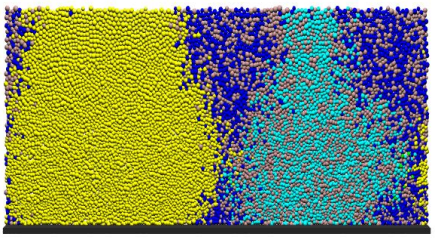


FIGURE 2.32: iDynoMICS graphics

This software tool allows to be extended because of the modularity that Java offers, so new ODE solvers and new functionalities (like the modelling of electrical fields or chemical reactions occurring separately from bacterial agent) can be added. It also uses XML inputs and outputs, which allows it to easily communicate with other software tools.

It is not implemented with parallelism, so as happens with the previous tools, a very big amount of individuals can not be simulated.

2.8.7 Summary

There are several biological processes which can be simulated by the five tools, but the remaining ones can only be handled by one or two simulators. Plasmid conjugation can only be simulated by DiSCUS, while EPS formation and excretion is only simulated by iDynoMICS. Another main difference between them is the computational performance, which can only considered very good in the case of CellModeller. A summary of the comparison between the five tools is showed in the following two tables:

Process	**DiSCUS**	**GRO**	**CellModeller**	**BactoSim**	**iDynoMICS**
Shape of the cell	✓	✓	✓	✗	✓
Elongation processes	✓	✓	✓	✓	✓
Cell division	✓	✓	✓	✓	✓
Pressure	✓	✓	✓	✓	✓
Viscous drag	✗	✗	✓	✗	✓
Growth speed	✓	✓	✓	✓	✓
Cell death	✗	✓	✗	✗	✓
Conjugation dynamics	✓	✗	✗	✓	✗
Intracellular processes	✓	✓	✓	✓	✓
Molecular signaling (Quorum sensing)	✗	✓	✓	✗	✓
Biofilms	✗	✓	✓	✗	✓
EPS	✗	✗	✗	✗	✓
Noise	✓	✓	✓	✓	✓
Physical media	✓Shape and size described with a Python function	✓Shape and size described with a GRO function	✓Shape and size described with a Python function	Shape and size described by Java code	Shape and size described by Java code
Chemostat mode	✗	✓	✗	✗	✓

Table 2.2: Simulated processes

Feature	**DiSCUS**	**GRO**	**CellModeller**	**BactoSim**	**iDynoMICS**
Based on	IBM	IBM	IBM	IBM	IBM
Computational performance	Bad, it needs parallel algorithms	Bad, only intracellular processes are parallel	Very good	Bad, it needs parallel algorithms	Bad, it needs parallel algorithms
Abstraction for intercellular processes	✗	✓	✓	✓	✓
Abstraction for intracellular processes	✓	✓	✓	✓	✓
Operating system	Multi-platform	Windows 7, Mac OS X	Multi-platform	Multi-platform	Multi-platform
Programming language	Python	GRO	Python	Java	Java
Representation of results	Pygame graphical library	Qt graphical library	Matplotlib graphical library	Repast GUI	Unknown
How to implement models	Prog. in Python	Prog. in GRO	Prog. in Python	Prog. in Java	Unknown
How to run it	Python line-commands	GRO graphical interface	Python line-commands	Repast GUI	Unknown
How easy to use it is for non-programmers	Very difficult	Difficult	Very difficult	Very difficult	Very difficult
How easy it is to extend	Very good	Good	Very good	Very good	Very good
Allows to define new types of cells/agents	✓	✓	✓	✓	✓
Independency between intracellular and intercellular processes	✓	✓	✓	✓	✓

Table 2.3: Simulators features

Chapter 3

Analysis of BactoSim I

3.1 User-level system analysis

In this section the system is described from a user point of view, simulating a bacterial model as an example and showing the main options of the graphical user interface. The best of the five compared simulation tools is CellModeller as it implements parallel computing, but it was just available for a small set of processors and graphical processing units, so BactoSim I was chosen, as it is the best option without taking CellModeller into account. It is more flexible as it is modified directly by programming in Java.

The figure 3.1 shows the main menu of the simulator. There are five buttons inside the red circle: load model, run model, run one step, stop and reset (from left to right). The green circle contains buttons which allow importing data to other useful tools like Weka, SQL, excel, R and so on.

There is also a screen in which the bacterial population and other selectable graphical charts are shown. To its left there are several tabs: the scenario tree, the simulation parameters and the run options.

The figure 3.2 shows the available menus of the top left. There are several modifiable parameters in the first screenshot, run scheduling options in the second one, and the scenario tree in the last one. Such three allows seeing and modifying the following files and options of the simulation:

- **Data Loaders:** specifies the class in which agents and layers are created.

- **Data Sets:** obtainable data from methods offered by several classes of the system.

- **Charts:** graphics in which data sets are shown and related.

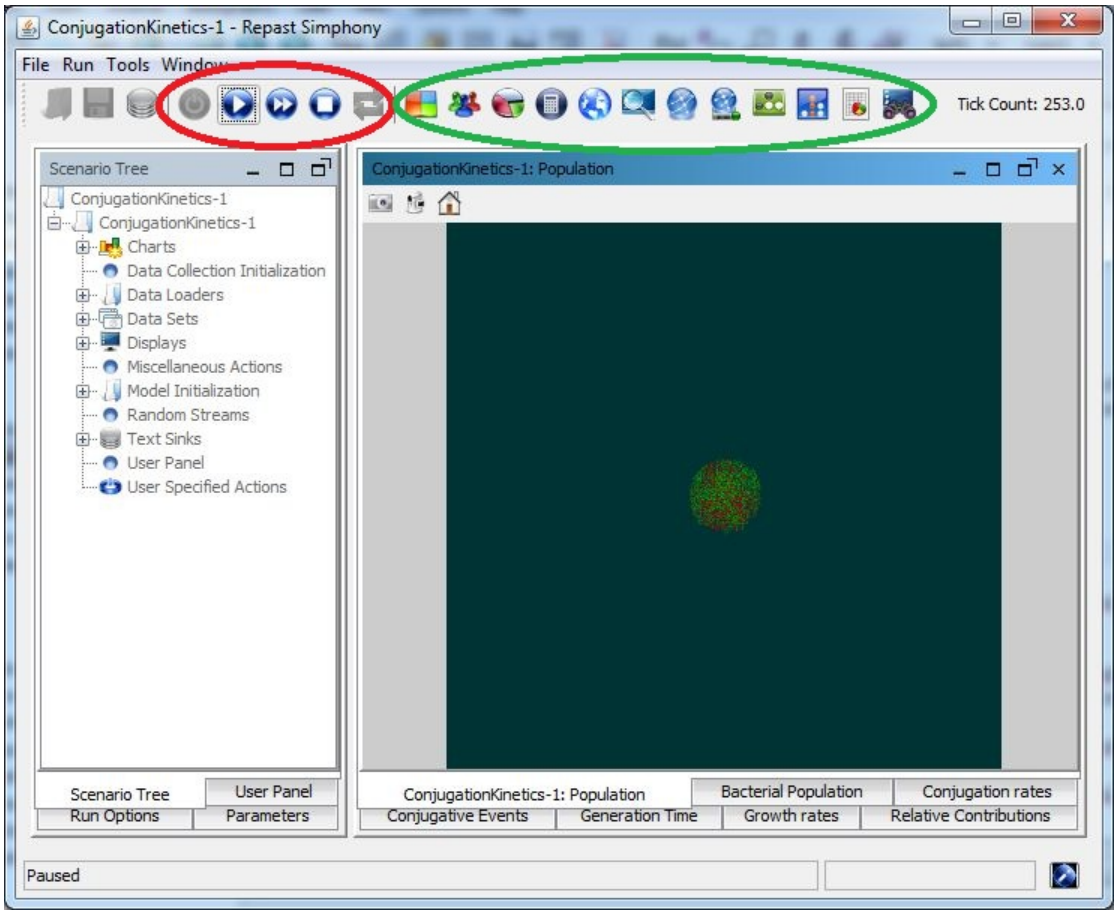


FIGURE 3.1: **Main menu of the simulator.**

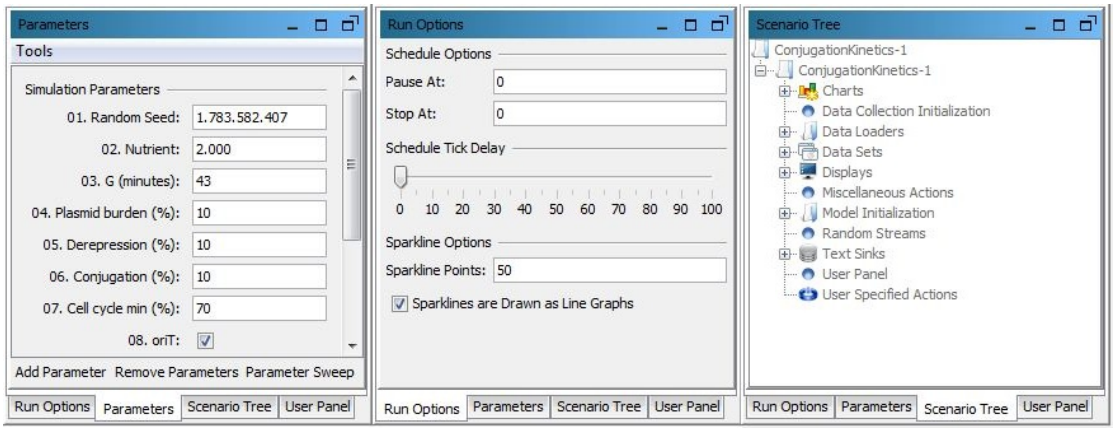


FIGURE 3.2: **Selectable menus of the simulator.**

- **Displays:** configurable screen in which interacting agents are visualised in 2D or 3D spaces.

- **Text Sinks:** files where data sets are gathered as text files with comma separated values format (csv files).

Figure 3.3 shows a bacterial population which grows when the user runs the simulation.

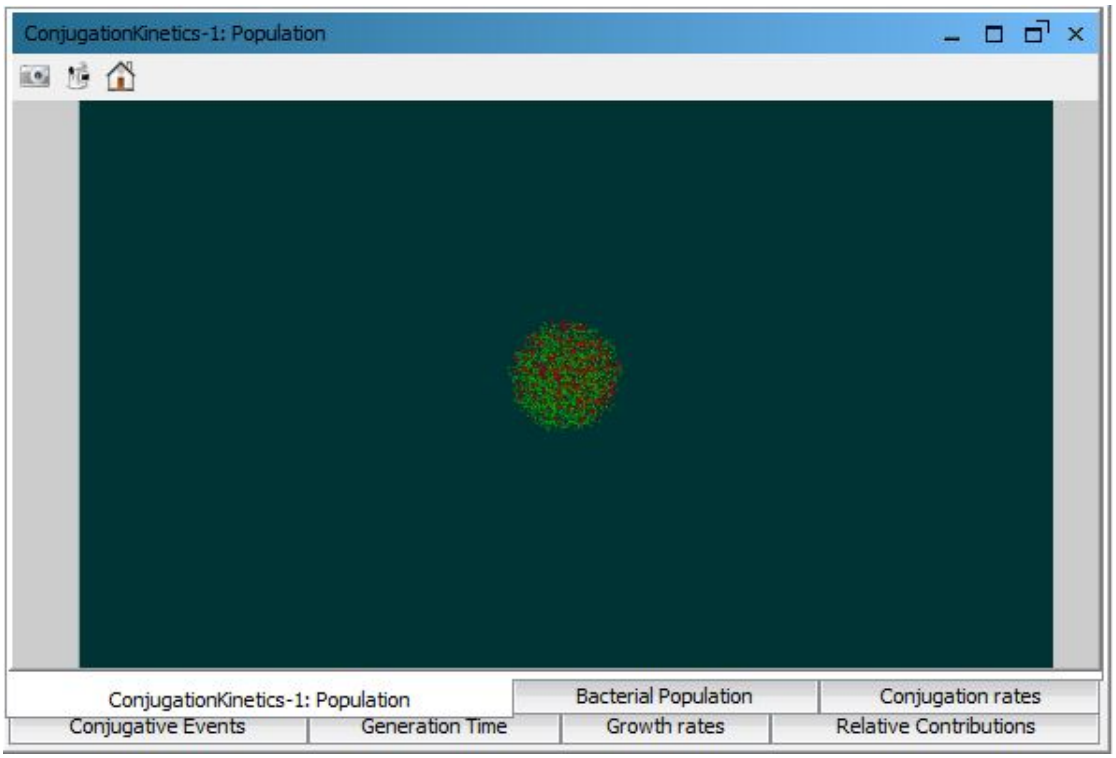


FIGURE 3.3: **Selectable menus of the simulator.**

Figure 3.4 shows a chart in which several census of such bacterial population are calculated over time. The other tabs have a similar appearance, but they show other data sets, like conjugative events, generation time, and so on.

3.2 Repast Symphony: a simulation system for Agent-Based Models

The simulator analysed is implemented using Repast Symphony 2.0 modelling system which can be integrated with eclipse by means of two different versions: ReLogo, which uses a programming language called Groovy that has some similarities with Java and generates the same bytecode, and Repast Java, which offers classes and methods to create a simulation environment using Java code.

This system conceptualizes the Agent-Based Models (ABM) as follows: it creates a **context** which can contain **individuals** and **layers**. Such layers contain points, its corresponding coordinates and values (which can be any type of object) associated with those points.

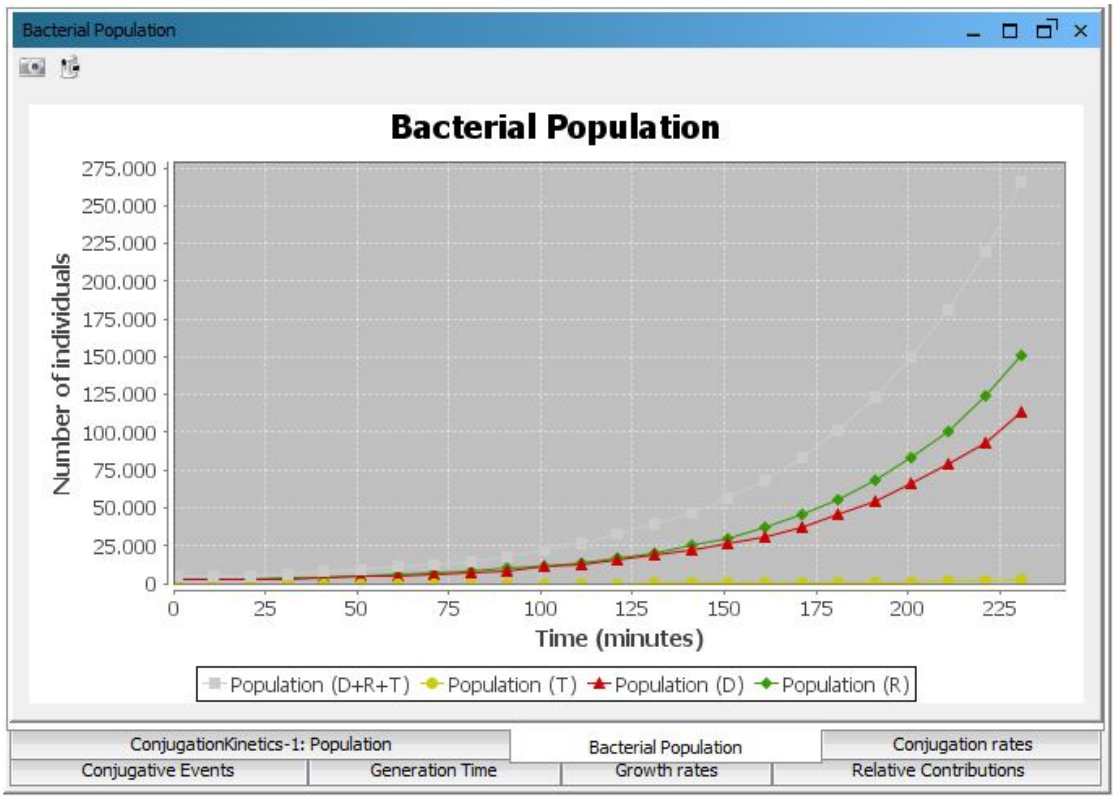


FIGURE 3.4: **Selectable menus of the simulator.**

A grid is a kind of layer which contains a cell in each coordinate and offers methods to compute the nearest neighbours implementing Moore's neighbourhood of size 1, 2, 3 and so on (figure 3.5[13]).

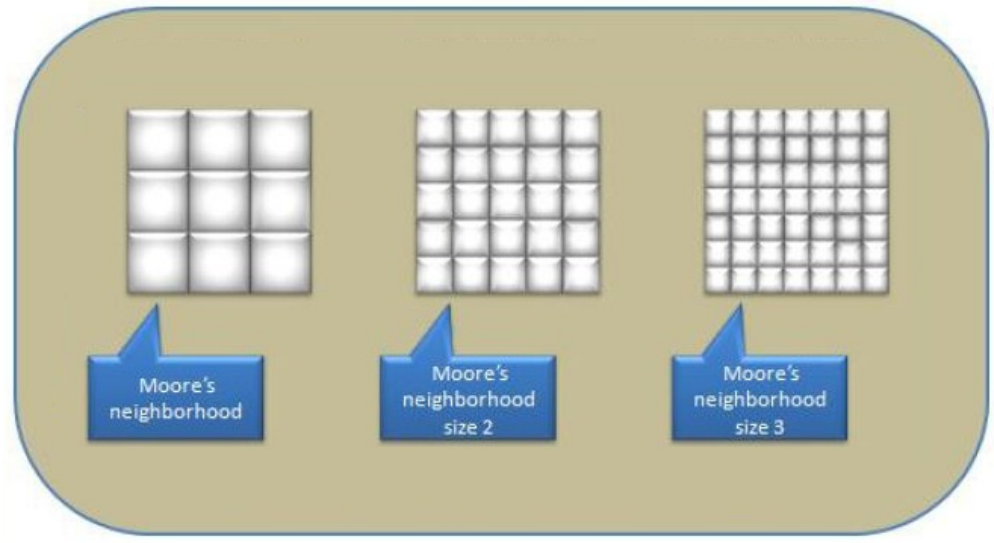


FIGURE 3.5: **The neighbourhood implemented in the framework.**

[13]Source: Prestes Garcia, Antonio. A first approach to individual-based modelling of the bacterial conjugation dynamics. Diss. Informatica, 2011.

All agents contained in the context can be associated to one or several layers simultaneously. This system assigns each agent to the *grid* to allow doing computations about their neighbouring agents and their displacements. Finally, there is also a nutrient layer to perform operations about nutrient availability for those agents which need this information. The architecture of the system is shown in the picture 3.6.

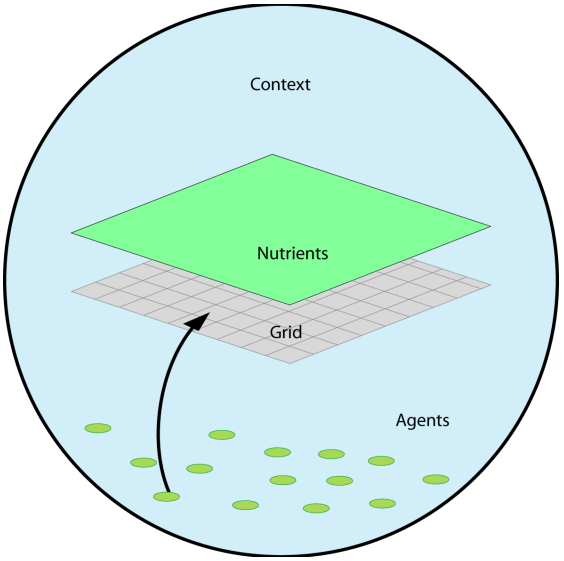


FIGURE 3.6: **Simulator system architecture.**

There is no need to create any class with a main method in order to run a Java project based on Repast Symphony. In fact, the basics to run an ABM are the following:

- Creating a class which implements the interface ContextBuilder and its method build(), where all agents and layers are built and associated between them.

- Creating an agent class which implements the step() method, which defines the actions performed by each individual at each instant of simulationtime.

Once the previous steps are done, the internal software of the system will call the build() method of ContextBuilder to set the simulation environment, and then it will call the step() method of each agent in each simulation time step, giving rise to a emergent behaviour from those individual behaviours.

Although those two types of classes are the only necessary, the analysed system includes more classes corresponding to certain parameters and computations needed in order to simulate this model, which are explained in the following section.

3.3 Class diagram and descriptions

3.3.1 Classes

This section shows the UML representation of the system (figure 3.7), as well as the following descriptions for each of the classes (see the appendix for a detailed description of each attribute and method of classes):

AbstractBacterium: Simple bacterial agent skeleton. It contains the attributes describing an abstract bacterium, its state, its properties and its conjugation configuration. It also contains getters and setters for its attributes, methods to keep track of the population and methods to modify several properties, all described below.

Bacterium: The bacterial cell agent implementation for model developed for FdlC group paper "Experimental validation of a kinetic numerical model of bacterial conjugation". The attributes are contained in the class it extends, i.e. AbstractBacterium. It contains methods to perform the nutrient uptake and diffusion, cellular division, conjugation processes and shoving forces.

BacteriumEquations: This class contains static methods used by the methods of the class Bacterium to calculate several characteristics of the cell, like cell mass, density, volume, length, nutrient uptake, growth, decay and some formulas to transform measuring units.

BacteriumStyle2D: This class corresponds to the implementation of a style interface to draw graphics with openGL. In this case, a shape factory is chosen to draw 2D shapes. Its methods are used to create the circular shape of bacteria and their colour, depending of their state (Donor = dark red, R = dark green, T = dark blue).

ModelRatesHelper: This class contains methods which return estimated rates for several processes: generation times, growth time or the conjugation rate. To do so, they use the end-point method among other probabilistic functions.

Note: This class uses the singleton pattern, i.e., the class allows to create only one instance of itself, because it contains a private constructor, a private static attribute.

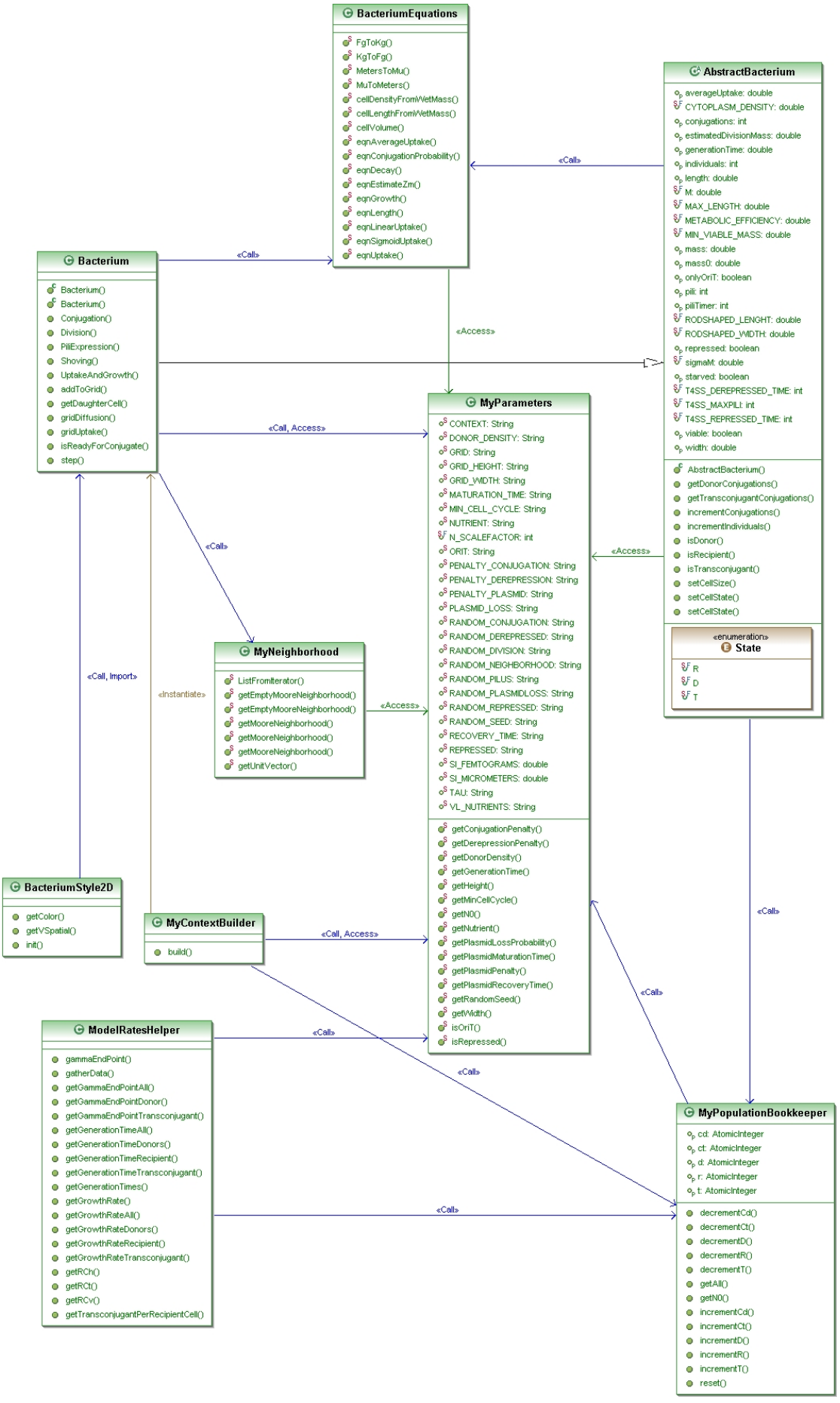


FIGURE 3.7: Class diagram

which is an object of the class itself and a method getInstance() which returns a reference to the only instance of the class.

MyContextBuilder: Class which builds a context by filling it with agents, adding projections and so on. This system does not have any "main" class with a "main" method, but this class could be considered the central part of the system, where the context, agents and layers are created.

MyNeighborhood: This class contains methods to return the moore's neighborhood grid points of variable size (1, 2, 3...) of a given grid point or agent. It also has methods to return the moore's neighborhood empty grid points of a given grid point or agent.

MyParameters: This class contain static attributes referencing string names of several required parameters, as well as their corresponding getters and setters.

MyPopulationBookkeeper: This is a data container class which stores and offers methods to modify the information related to the number of donor, recipent and transconjugant cells, as well as the conjugative transfers performed by Donor and Transconjugant cells

3.3.2 Datasources

This folder contains several classes which are the source for aggregate data shown in the Graphical User Interface:

- **CFreqEndpoint:** Gamma end-point (D+R+T growth rate).

- **CFreqEndpointDonor:** Gamma end-point (D growth rate).

- **CFreqEndpointTransconjugant:** Gamma end-point (T growth rate).

- **CFreqTtoRT:** Rate (T/(R+T)).

- **GenerationTimeAll:** Generation Time G(D+T+R).

- **GenerationTimeD:** Generation Time G(D).

- **GenerationTimeR:** Generation Time G(R).

- **GenerationTimeT:** Generation Time G(T).

- **GrowthRateAll:** Population growth rate.

- **GrowthRateDonors:** Donors growth rate.

- **GrowthRateRecipients:** Recipients growth rate.

- **GrowthRateTransconjugants:** Transconjugants growth rate.

- **PopulationAll:** Population (D+R+T).

- **PopulationDonors:** Population (D).

- **PopulationRecipient:** Population (R).

- **PopulationTransconjugant:** Population (T).

- **RCh:** Relative contribution of horizontal transfer.

- **RCt:** Relative contribution of transconjugants re-transfers.

- **RCv:** Relative contribution of vertical transfer.

They all are implementations of the interface AggregateDataSource, which extends the interface DataSource, and they all are based on the folling code:

```
import org.holistic.bactocom.ModelRatesHelper;
import repast.simphony.context.Context;
import repast.simphony.data2.AggregateDataSource;

public class <<name of the class>> implements AggregateDataSource {

        /**
         * Gets the unique id of this DataSource.
         */
        @Override
        public String getId() {
                return "<<name of the data>>";
        }

        /**
         * Gets the type of data produced by this DataSource.
         */
        @Override
        public Class<?> getDataType() {
                return int.class;
        }

        /**
         * Gets the type of the object that this DataSource can retreive data from.
         */
        @Override
        public Class<?> getSourceType() {
                return Context.class;
        }

        /**
         * Gets the data using the specified iterable.
```

```
         *
         * @param size  The number of objects in the iterable.
         * @param objs The iterable over objects to use in getting the data.
         * @return the data using the specified iterable.
         */
        @SuppressWarnings("rawtypes")
        @Override
        public Object get(Iterable<?> objs, int size) {
                <<statements>>
                return <<Data to return>>;
        }


        /**
         * Resets this AggregateDataSource prior to the next get call.
         */
        @Override
        public void reset() {
        }

}
```

3.4 XML as GUI descriptor

Data shown in the graphical interface can be modified by means of the Scenario Tree or editing the XML files contained in the folder "ConjugationKinetics2D-v1.rs" of the project. The last option is more flexible, so it will be the one used for this purpose.

As seen in the user-level analysis, Scenario Tree shows several options: Data Loaders, Data Sets, Charts, Displays and Text Sinks. Such options are described by several XML files, which are shown below:

Data Loaders:

- repast.simphony.dataLoader.engine.ClassNameDataLoaderAction_0.xml

Data Sets:

- repast.simphony.action.data_set_1.xml

- repast.simphony.action.data_set_2.xml

- repast.simphony.action.data_set_3.xml

- repast.simphony.action.data_set_4.xml

- repast.simphony.action.data_set_5.xml

- repast.simphony.action.data_set_6.xml

Charts:

- repast.simphony.action.time_series_chart_15.xml

- repast.simphony.action.time_series_chart_16.xml

- repast.simphony.action.time_series_chart_17.xml

- repast.simphony.action.time_series_chart_18.xml

- repast.simphony.action.time_series_chart_19.xml

- repast.simphony.action.time_series_chart_20.xml

Displays:

- repast.simphony.action.display_14.xml

Data Sinks:

- repast.simphony.action.file_sink_7.xml

- repast.simphony.action.file_sink_8.xml

- repast.simphony.action.file_sink_9.xml

- repast.simphony.action.file_sink_10.xml

- repast.simphony.action.file_sink_11.xml

- repast.simphony.action.file_sink_12.xml

- repast.simphony.action.file_sink_13.xml

The tags $< entry >$ of each file contain the modifiable fields, and new entries can also be added. For example, in the population chart (figure 3.8) there are data about D, T, R and D+R+T population census. The XML file repast.simphony.action. time_series_chart_17.xml is shown in the figure 3.9.

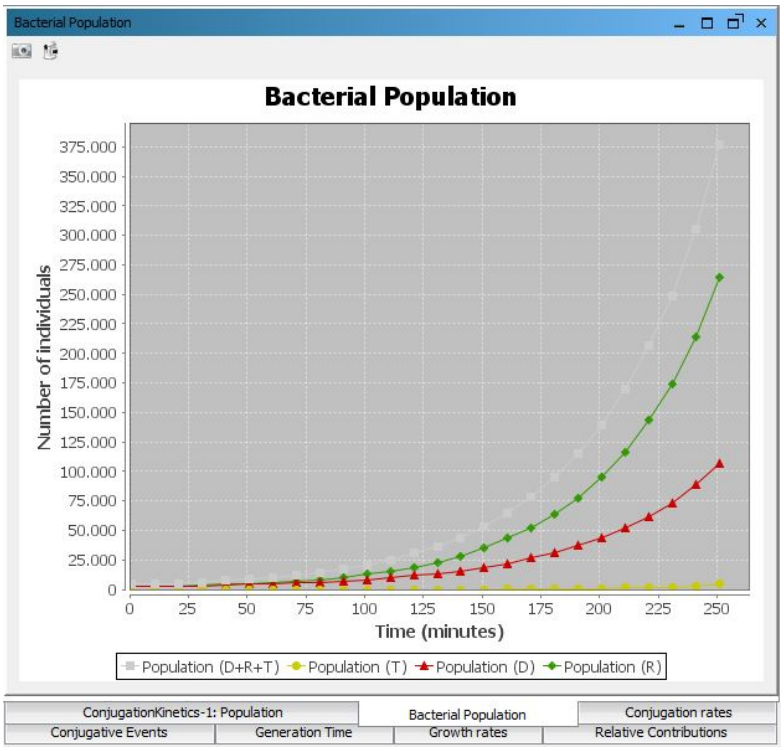


FIGURE 3.8: **Screenshot of the population chart.**

What about if showing the population census of R+T is also needed?

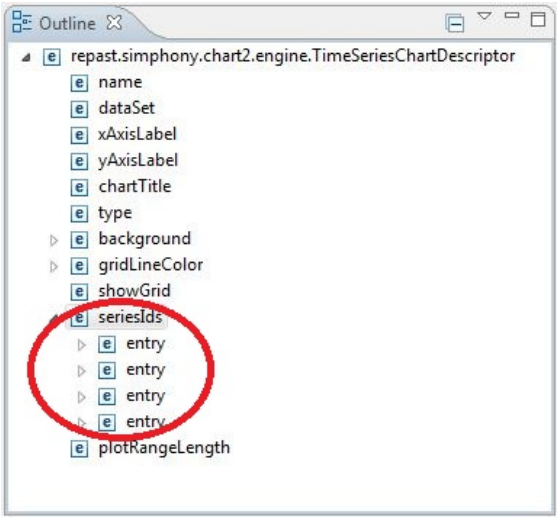


FIGURE 3.9: **Screenshot of the population XML file structure.**

A new entry with name and label $Population(R+T)$ has to be added, specifying the desired color, and the Data Set from which this chart gathers data has to be modified as this XML only defines what is shown in the chart. The following tag has to be observed for that purpose:

$< dataSet > PopulationCensus < /dataSet >$

And its corresponding XML file repast.simphony.action.data_set_3.xml (PopulationCensus), from whose structure is shown in the figure 3.10, has to be modified.

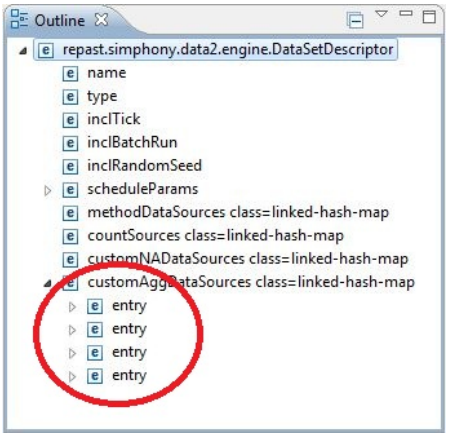


FIGURE 3.10: **Screenshot of the population XML data set file structure.**

For that purpose, its name, id and the name of the class which implements the "AggregateDataSource" interface and from which such data are taken, i.e. "org.holistic.bactocom.datasources.PopulationRT", have to be specified. Such class does not exist, so a new class called PopulationRT which implements the interface "AggregateDataSource" has to be created in the package "datasources" of the project, as it is shown in the code snipet 3.1.

```
package org.holistic.bactocom.datasources;

import org.holistic.bactocom.MyPopulationBookkeeper;

import repast.simphony.context.Context;
import repast.simphony.data2.AggregateDataSource;

public class PopulationRT implements AggregateDataSource {

        @Override
        public String getId() {
                return "Population (R+T)";
        }

        @Override
        public Class<?> getDataType() {
                return int.class;
        }

        @Override
        public Class<?> getSourceType() {
                return Context.class;
        }

        @Override
        public Object get(Iterable<?> objs, int size) {
```

```
                double p= (double) MyPopulationBookkeeper.getInstance().getRT();
                return p;
        }

        @Override
        public void reset() {}

}
```

LISTING 3.1: lst:PopulationRT

The method *get*() of this new class contains a call to the method "MyPopulationBookkeeper . getInstance() . getRT()" which does not exist yet, so it has to be defined in the class "MyPopulationBookkeeper", as shown in the code snipet 3.2.

```
        public double getRT() {
                return(getR() + getT());
        }
```

LISTING 3.2: lst:getRT

When those previous steps are finished, the Java code of the simulation tool has to be compiled and executed, so a new graphic can be visualised (figure 3.11).

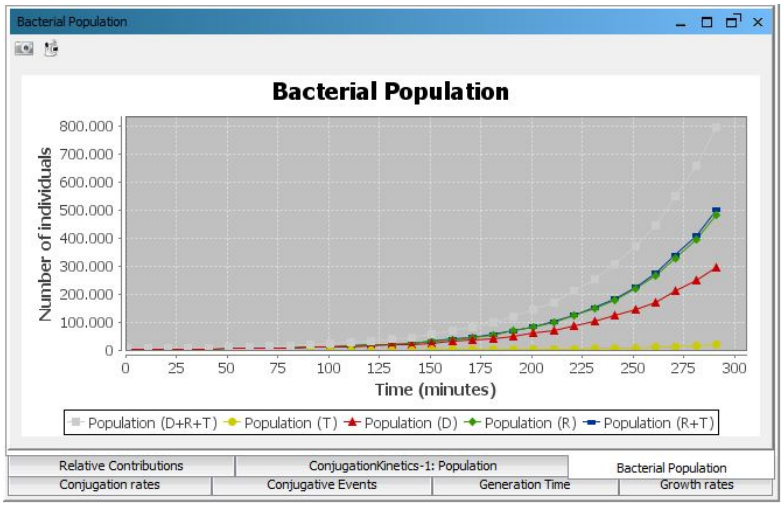


FIGURE 3.11: **Screenshot of the population chart with a new graphic.**

It would not be necessary to delete all the new code of the classes and XML files if the changes would have to be undone, because the graphical user interface offers an easy way to do it, disabling the new graphic as it is shown in figure 3.12.

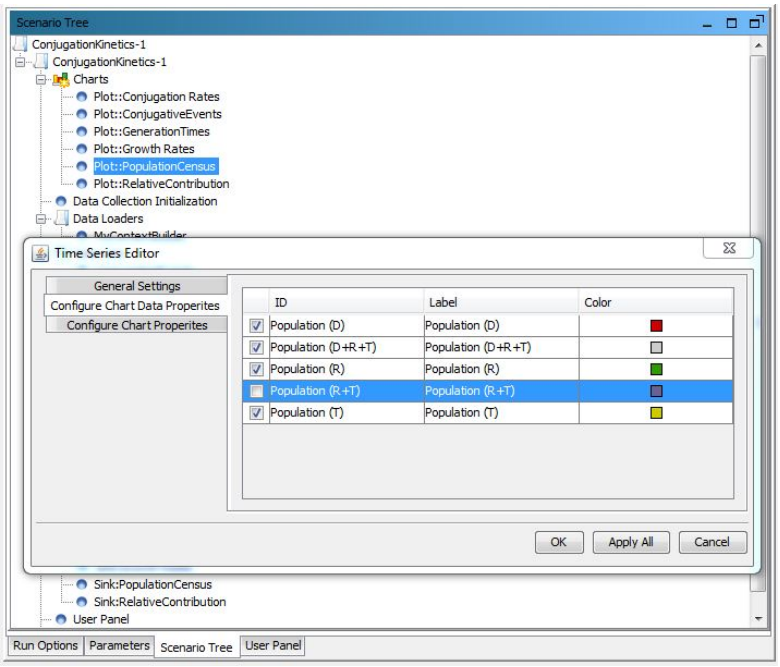


FIGURE 3.12: **How to disable the new graphic.**

The previous example showed how to describe the graphical user interface by means of XML files and how to add a new graphic for a new Data Set. This action will be needed when adding information about the infection process with phages.

Chapter 4

Design and implementation of the new functionality

4.1 Design of the infection process

Based on the theory gathered in chapter 2 (State of the Art), the interaction between bacteriophages and bacteria is based in the absorption, release and diffusion of such viruses, processes modelled by the following mathematical formulas:

- **Local MOI (Local Multiplicity of Infection):** Determines the number of phages that infect the cell.

$$LocalMOI(ts) = ml_{ts} = \frac{\text{phages in radius 1 from the studied cell}}{\text{cells in radius 2 from the studied cell}}$$

- **Probability of infection (local):** Generating a random number and comparing it to this value, it can be determined if a single bacterium will be infected by $LocalMOI$ viruses in one time step:

$$P_{inf-local} = P(K >= 1) = 1 - P(K = 0) = 1 - e^{-ml/ts}$$

where ts (search time) is a parameter which can be set by the user.

- **Eclipse time:** external parameter which can be set by the user. It determines the time required for phages to replicate into bacteria and be released.
- **Release of phages:** this process depends of an infection clock which is activated in each bacteria when it is first infected by a bacteriophage, and it is incremented by

one in every time step. When such infection clock is bigger than the eclipse time, the bacterium starts releasing phages following this cycle:

1 phage released, **1** phage released, **2** phages released, **1** phage, **1** phage, **2** phages, ... where each time step is separated by commas.

If a cell divides, its infection clock is also divided by 2.

- **Diffusion of phages:** viruses move within the agar gel following this formula:

$$d = \sqrt{2 * 2 * D * 60} * 10^6 = 2 * 10^6 * \sqrt{60 * D}$$

where d is the distance (in grid cells) that the phage randomly moves.

All this new functionality has been implemented, modifying the previous software, which was only able to simulate the exchange of plasmids between bacteria.

4.2 Extended class diagram and descriptions

In this section, two subsets of the final class diagram are shown to explain the main changes made to the previous software tool. Figure 4.1 shows the new classes corresponding to the phage agents and how they interact with the previous classes. PhageFactory implements the Single Factory Pattern, which allows the client creating different types of phages without knowing anything about the logic involved in their instantiation, i.e., clients only use the IPhage interface to deal with the viruses. AbstractPhage is an abstract class used to implement the common behaviour of each type of phage, i.e. the $diffusion()$ method. Bacterium class shows the new implement methods $absorbtionOfPhages()$ and $releaseOfPhages()$, corresponding to the new functionality.

In the other hand, figure 4.2 shows the new attributes and methods implemented in other classes in order to extend the functionality of the simulation tool. Such extensions are explained below:

- **AbstractBacterium:** this class includes new attributes related to the infection clock, the strength of the promoter of the first bacteriophage which infects the bacterium, which will capture all the machinery of the cell for its replication purposes, and the type of phage which has first infected the cell. The cell may absorb more phages, but this simplified model does not store information about other types of infection, as the first

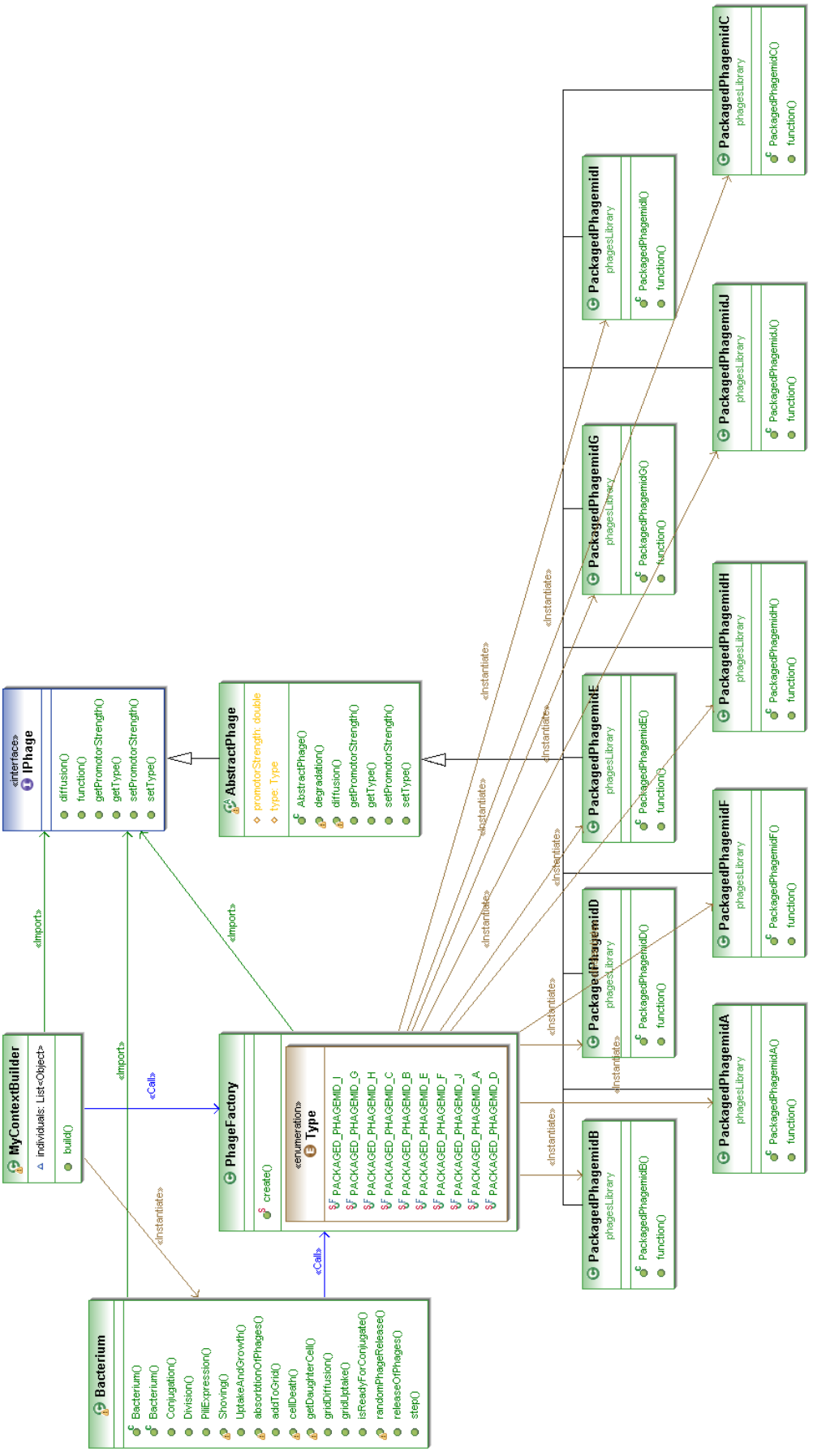


FIGURE 4.1: **subset 1 of the final class diagram**

FIGURE 4.2: **subset 2 of the final class diagram**

phage which infects a cell is the only one replicating into such cell. It also includes two new states to its enumerate State, as bacteria can be infected or uninfected.

- **MyParameters:** this class now includes new attributes and its corresponding getters and setters for the new parameters included in the Graphical User Interface. Such new parameters are explained in the next section.

- **MyPopulationBookkeeper:** this class keeps track of the amount of individuals to display that information in the Graphical User Interface. It includes now attributes to keep track of the amount of phages and its corresponding enhancers.

- **ModelRatesHelper:** this class reads the current data taken from the aggregate data sources, so it can send information to MyPopulationBookkeeper, which directly calls this class to take information about the population. It includes now new methods to send information about all types of phages.

4.3 GUI extension using XML files

Graphical User Interface has been extended following the steps explained in chapter 3:

- New parameters have been included, while others have been deleted, as it is shown in figure 4.3.

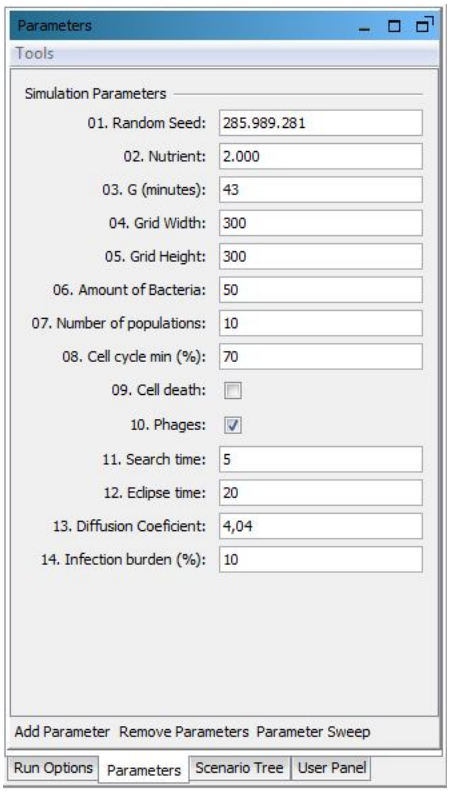


FIGURE 4.3: **New set of parameters**

- Population chart shows bacteriophages as blue, small circles (figure 5.2).

- Bacterial Population chart shows infected and uninfected bacteria, instead of donors, recipients or transconjugants bacteria (figure 5.3).

- Viral Population is a new chart which shows the amount of the different types of bacteriophages over time (figure 5.4).

Chapter 5

System Evaluation

5.1 Code Verification

Some of the new methods with their attributes have been tested using simple black box test cases, although most of the methods could not be tested in isolation, as their execution depends directly on the internal classes of Repast Symphony. Anyway, as they are implemented with existing methods of Repast Symphony, most of the testing has been based on the experimental validation. For example, if there was an error or an exception in the code of a single method, it was straightforward to find its corresponding error in the simulation.

The way to test methods using experimental validation was developing increasingly complex methods. For example, the method $absorbtionOfPhages()$ in $Bacterium$ was first developed to absorb just one phage contained in its grid cell. The second step was making it able to absorb a phage in the neighbouring cells. The last step was making it able to absorb two phages and choose one of them randomly. After that, the absorption of several phages around the cell was easy to implement.

Other methods, as those corresponding to the data shown in the Graphical User Interface were easily tested just by observing the charts of the simulator.

5.2 Model Fitting and Experimental Validation

After verifying each new method of the code, an experiment of directed evolution has to be simulated to check if the implemented model fits the data obtained from experimental results. In this case, the simulated experiment consists of ten bacterial colonies, each of them contains a different type of bacteriophage M13 (figure 5.1). The difference between phages consists of their promotor strength.

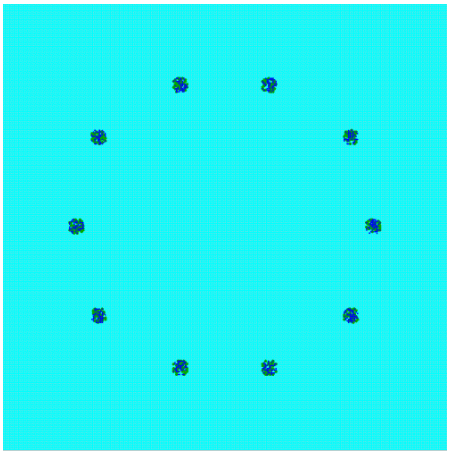


FIGURE 5.1: **Ten bacterial populations**: each of them containing a different type of phage(blue).

All bacteria contained in every bacterial colonies are identical, they only vary in their mass as it is randomly chosen when they are created by the context builder of the system. The real experiment would consists of bacteria containing a transcription factor (TF) which binds to the promoter of the phagemid vectors contained into the bacterium (when phages have infected the bacterium with their single stranded DNA sequence, i.e. the phagemid vector). When such TF binds to the phagemid vector, it is replicated, so new progeny is being created inside the bacterium. If the promoter is strong, the TF will bind to it more easily than a weak promoter, so the phagemid vector will replicate more times than one containing a weak promoter. Different phagemids containing different promoter strengths can be obtained by creating a mutant library.

In order to simulate the promoter strength in each of the ten colonies, an uniform distribution has been implemented in the phage library, so that each phage has a different strength from 0.1 up to 1. This is an attribute called "*double promotorStength*" which is directly used in the $releaseOfPhages()$ method to calculate the probability of a bacterium releasing phages in each time step.

This experiment simulates a parallel search of the phagemid vector containing the strongest promoter, and the results are shown in the pictures below. They prove that the amount of released phages vary depending of the promoter strength directly.

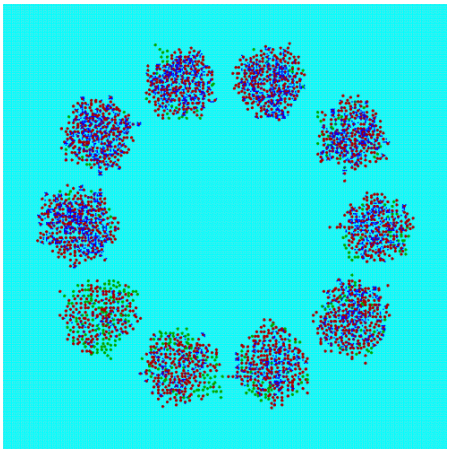


FIGURE 5.2: **Ten bacterial populations**

Picture 5.2 shows how some bacterial populations are more infected (red bacteria) than others. The amount of bacteriophages also vary depending on the promotor strength. The first colony (the top left one) has a stronger promoter, and their strengths decreases in each colony observing them sequentially as clockwise.

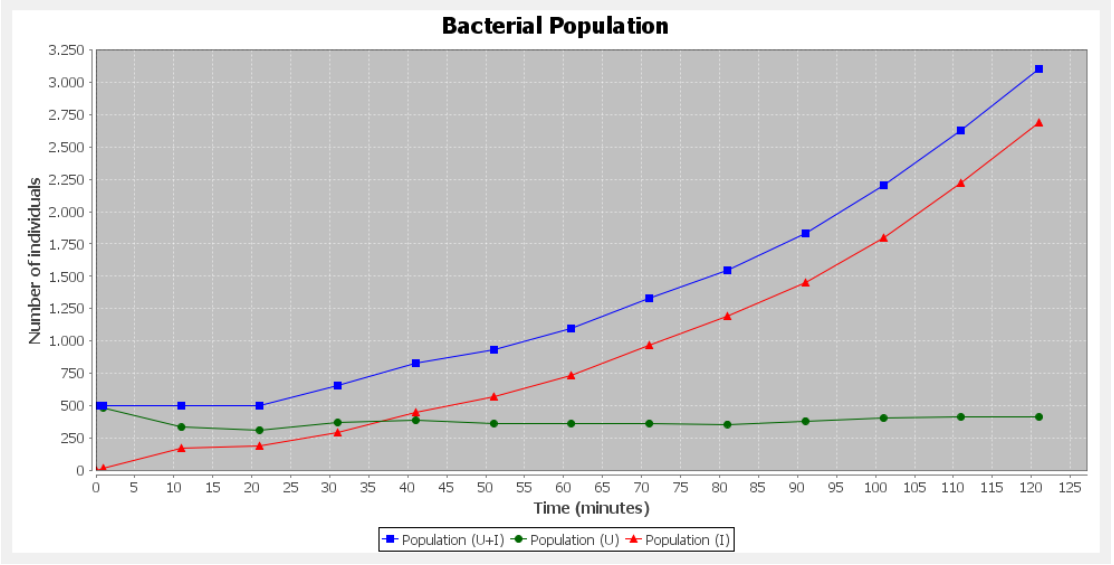


FIGURE 5.3: **Amount of infected and uninfected bacteria**

Picture 5.3 shows how the amount of uninfected bacteria decreases over time as new phages are released, infecting more bacteria over time.

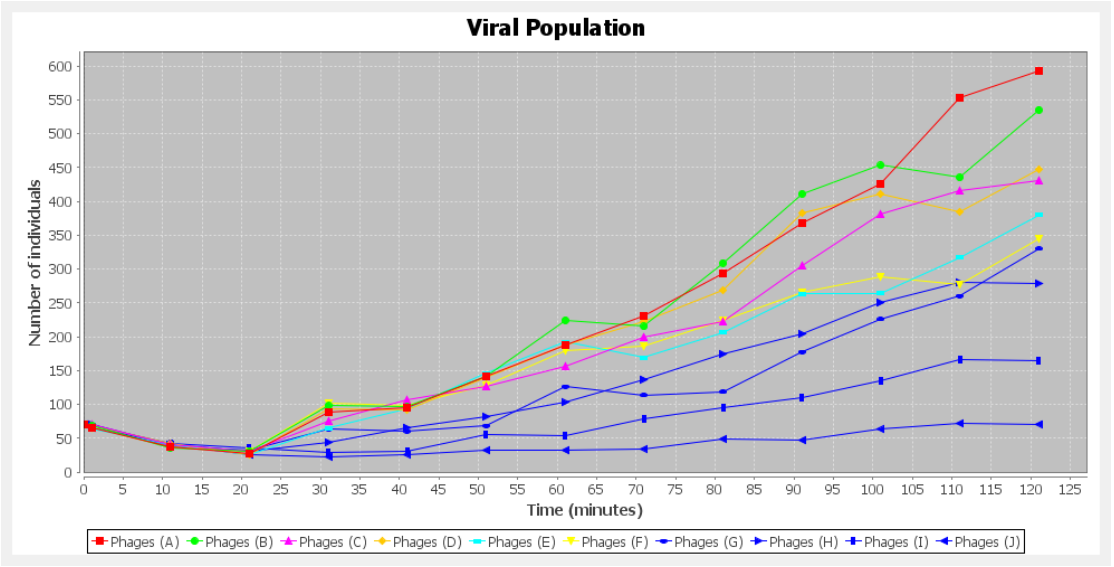


FIGURE 5.4: **Amount of the ten different types of phages**

Picture 5.4 shows how the amount of each type of bacteriophage depends directly on the strength of its promoter. Phage A has a stronger promoter than Phage B, Phage B has a stronger promoter than Phage C, and so on.

Chapter 6

Conclusions and future work

In this project, the interaction between bacteriophages M13 and Escherichia Coli has been simulated carrying out an experiment of directed evolution based on the parallel search of the stronger phage, starting from a library of phages which already contains the solution. It could be considered a different version of the PACE method which implements a genetic algorithm as a single population of phages is evaluated and evolved in each cycle of the process, giving rise to a final population of phages which contains a strong promoter [19]. This simulation showed results which fit with experimental data. During the execution, the software had more difficulties when simulating big amounts of bacteria and phages, as the number of individuals increased exponentially. Those difficulties were easily appreciated after several tick counts, as the simulation ran slower. This fact shows that parallelism could improve the simulation results, so a new simulation tool based on the same model could be developed and implemented using parallel computing.

In the other hand, this simulation tool allows simulating a simplified version of bacteria and phages. The research group in which this project has taken place is actually developing new computational models to be implemented in bacteria and phages directly, so the implemented code of such individuals could be improved to simulate computations performed between them. Thus, when the simulation results of such biological computations are reliable, a real implementation of them could be accomplished in a biotechnology laboratory, giving rise to new bacterial computing models based on remote Horizontal Gene Transfer using bacteriophages M13.

Chapter 7

Appendix

7.1 Detailed descriptions of classes

7.1.1 AbstractBacterium

Attributes:

State	
Data type: public static enum	**Initial value:** { D, R, T }
The state of the bacterium, which can be { D, R, T }, where: D is donor, T is transconjugant, R is recipient.	

RODSHAPED_WIDTH	
Data type: public final static double	**Initial value:** 0.5
Standard width for rod-shaped bacteria.	

RODSHAPED_LENGHT	
Data type: public final static double	**Initial value:** 0.6
Initial length value for rod-shaped bacteria.	

METABOLIC_EFFICIENCY	
Data type: public final static double	**Initial value:** 0.6
Amount of up-take converted in cell mass (0.5-0.6).	

T4SS_MAXPILI	
Data type: public final static int	**Initial value:** 5
Average pili number per cell (E. coli 4-5).	

T4SS_REPRESSED_TIME	
Data type: public final static int	**Initial value:** 1000
Time in which T4SS genes are repressed.	

T4SS_DEREPRESSED_TIME	
Data type: public final static int	**Initial value:** 40
Time in which T4SS genes are repressed.	

MIN_VIABLE_MASS	
Data type: public final static double	**Initial value:** 120
Minimum viable mass for cells.	

MAX_LENGTH	
Data type: public final static double	**Initial value:** 4
Maximal allowed rod-shaped length.	

CYTOPLASM_DENSITY	
Data type: public final static double	**Initial value:** 1200
Average cell density in kg/m^3.	

M	
Data type: public final static double	**Initial value:** 1200
Average cell mass at division.	

sigmaM	
Data type: public final static double	**Initial value:** 120
Standard deviation of cell mass at division.	

averageUptake	
Data type: private double	**Initial value:** Double. NaN
Average nutrient uptake.	

estimatedDivisionMass	
Data type: private double	**Initial value:** Double. NaN
Division mass.	

individuals	
Data type: private int	**Initial value:** 1
Number of individuals which this agent (super-individual) represents.	

*mass*0	
Data type: private double	**Initial value:** Double.NaN
The initial cell mass at t=0 in femtograms.	

mass	
Data type: private double	**Initial value:** Double.NaN
The current cell mass in femtograms.	

length	
Data type: private double	**Initial value:** No initialized
The length of cell from pole to pole.	

width	
Data type: private double	**Initial value:** No initialized
The width of cell. For rod shaped bacteria this is a fixed value of 0.5 micrometers.	

*state*0	
Data type: private State	**Initial value:** null
The initial cell state.	

state	
Data type: private State	**Initial value:** null
The current cell state.	

pili	
Data type: private int	**Initial value:** 0
The number of conjugative pili on cell surface.	

piliTimer	
Data type: private int	**Initial value:** 0
Time required to express the conjugative machinery.	

conjugations	
Data type: private int	**Initial value:** 0
Number of conjugative events performed by a single cell.	

starved	
Data type: private boolean	**Initial value:** false
Cell is no longer growing due to nutrient exhaustion.	

viable	
Data type: private boolean	**Initial value:** true
Cell is no longer viable.	

generationTime	
Data type: private double	**Initial value:** 0
The bacterial generetion time G.	

onlyOriT	
Data type: private boolean	**Initial value:** false
If true is a mobilizable non-conjugative plasmid.	

repressed	
Data type: private boolean	**Initial value:** false
Tells if plasmid is repressed.	

Methods:

AbstractBacterium (double m, State s, double G, boolean R, boolean b)
Type: public AbstractBacterium
Inputs: - *m:* The initial cell mass - *s:* The initial cell state - *G:* The Generation Time - *R:* A boolean flag indicating the plasmid has the Tra genes repressed - *b:* True if plasmid is a mobilizable plasmid
Description: The abstract constructor.

getState0 ()
Type: public State
Inputs: No inputs
Description: return the initial cell state.

setState0 (State s)
Type: public void
Inputs: - **state0:** the state0 to set
Description: sets the initial cell state.

getState ()
Type: public State
Inputs: No inputs
Description: Returns the current cell state.

setState (State s)
Type: public void
Inputs: - *state*: the state to set
Description: Sets the current cell state.

populationAccount (State s0, State s)
Type: private void
Inputs: - *s0*: The cell state at t0 - *s*: The cell state at current time
Description: This method keeps the track of number and the type of individuals in simulated population. Given a state s (D, R or T), *MyPopulationBookkeeper* class will increment such state. If the initial and the current state are the same, the method stops here. Given an initial state s0, *MyPopulationBookkeeper* class will decrement such state.

getMass0 ()
Type: public State
Inputs: No inputs.
Description: Returns the initial cell mass.

setMass0 (double m)
Type: public void
Inputs: - m: the mass to set
Description: Sets the initial cell mass.

getMass ()
Type: public State
Inputs: No inputs
Description: Returns the current cell mass.

setMass (double m)
Type: public void
Inputs: - m: The mass to set
Description: Sets the current cell mass.

getIndividuals ()
Type: public int
Inputs: No inputs
Description: Return The actual number of real bacterial cells. In super-individual scheme every agent in the computational domain stands for several real individuals in the concrete domain.

setIndividuals (int v)
Type: public void
Inputs: - *v:* Number of individuals
Description: Sets the number of individuals.

incrementIndividuals ()
Type: public void
Inputs: No inputs
Description: Increments the variable *individuals* in one unit.

getConjugations ()
Type: public int
Inputs: No inputs
Description: Returns the number of conjugative events performed by each cell, i.e., returns the attribute *conjugations*.

setConjugations (int v)
Type: public void
Inputs: - v: Number of conjugative events
Description: Sets the value of conjugative events which cell had performed.

incrementConjugations ()
Type: public void
Inputs: No inputs
Description: Increment the number of conjugative events, i.e., increments the attribute *conjugations* and *MyPopulationBookkeeper* class increments its Cd or Ct attributes which correspond to the number of conjugative transfers performed by Donor cells (if the bacterium is donor) or Transconjugant cells (if the bacterium is transconjugant), respectively.

getLength ()
Type: public double
Inputs: No inputs
Description: Returns the *length* of the bacterium.

setLength (double length)
Type: public void
Inputs: - *length:* the length to set
Description: Sets the *length* of the bacterium.

getWidth ()
Type: public double
Inputs: No inputs
Description: Returns the *width* of the bacterium.

setWidth (double width)
Type: public void
Inputs: - *width:* the width to set
Description: Sets the *width* of the bacterium.

setCellSize (double w, double l)
Type: public void
Inputs: - w: The cell width - l: The cell length
Description: A simple wrapper to set both cell dimensions.

setCellState (double m, double w, double l, State s,int p, int t)
Type: public void
Inputs: - m: The cell mass - w: The cell width - l: The cell length - s: The cell state - p: The number of conjugative pili on cell surface - t: The timer for the expression of a new pilus
Description: A simple wrapper to set several cell state variables at once. It sets the *mass* to the given mass, the cell size based on the current mass and the value of the constant $CYTOPLASM\_DENSITY$, the number of conjugative pili in the cell surface, the timer for the expression of a new pilus and the cell state.

setCellState (double m, State s, int p)
Type: public void
Inputs: - m: The cell mass - s: The cell state - p: The number of conjugative pili on cell surface
Description: A simple wrapper to set several cell state variables at once. It sets the *mass* to the given mass, the cell size based on the current mass and the value of the constant $CYTOPLASM\_DENSITY$, the number of conjugative pili in the cell surface and the cell state. The difference between the previous method is this one doesn't set the pili timer.

isStarved ()
Type: public boolean
Inputs: No inputs
Description: Returns the *starved* attribute, indicating if the cell is starving or not.

setStarved ()
Type: public void
Inputs: - *starved*: the starved to set
Description: Sets the *starved* attribute, indicating if the cell is starving or not.

isViable ()
Type: public boolean
Inputs: No inputs
Description: If the current mass of the cell is smaller than the constant MIN_VIABLE_MASS, it sets the attribute *viable* to false. Returns such attribute, indicating if the cell is no longer viable.

setViable ()
Type: public void
Inputs: - *v*: the viability state
Description: Sets the *viable* attribute, indicating if the cell is no longer viable.

getPili ()
Type: public int
Inputs: No inputs
Description: Returns the number of *pili* in the cell surface.

setPili (int pili)
Type: public void
Inputs: - *pili*: the pili to set
Description: Sets the number of *pili* in the cell surface.

isDonor ()
Type: public boolean
Inputs: No inputs
Description: Helper method to evaluate the cell state. Returns true if this bacterium is a plasmid Donor.

isRecipient ()
Type: public boolean
Inputs: No inputs
Description: Helper method to evaluate the cell state. Returns true if this is a plasmid Recipient.

isTransconjugant ()
Type: public boolean
Inputs: No inputs
Description: Helper method to evaluate the cell state. Returns true if this is a plasmid Transconjugant.

getDonorConjugations ()
Type: public int
Inputs: No inputs
Description: If the bacterium is donor, this method returns the attribute *conjugations*. If it is not donor, the method returns 0.

getTransconjugantConjugations ()
Type: public int
Inputs: No inputs
Description: If the bacterium is transconjugant, this method returns the attribute *conjugations*. If it is not transconjugant, the method returns 0.

getAverageUptake ()
Type: public double
Inputs: No inputs
Description: Returns the *averageUptake* attribute, which indicates the average nutrient uptake.

setAverageUptake (double v)
Type: public void
Inputs: - *v*: the averageUptake to set
Description: Sets the *averageUptake* attribute.

getEstimatedDivisionMass ()
Type: public double
No inputs
Description: Returns the *estimatedDivisionMass* attribute.

setEstimatedDivisionMass (double v)
Type: public void
Inputs: - *v*: the estimatedDivisionMass to set
Description: If the value of *estimatedDivisionMass* is not NaN, the method sets it.

getGenerationTime ()
Type: public double
Inputs: No inputs
Description: Returns the attribute *generationTime*, indicating the required time for baterial generation.

setGenerationTime (double v)
Type: public void
Inputs: - v: the generationTime to set
Description: Sets the attribute *generationTime*.

isOnlyOriT ()
Type: public boolean
Inputs: No inputs
Description: Returns the *onlyOriT* attribute, indicating if the bacterium contains a mobilizable non-conjugative plasmid.

setOnlyOriT (boolean v)
Type: public void
Inputs: - v: the onlyOriT to set
Description: Sets the *onlyOriT* attribute.

isRepressed ()
Type: public boolean
Inputs: No inputs
Description: Returns true if the plasmid is repressed, false if it is not repressed.

setRepressed (boolean v)
Type: public void
Inputs: - v: the repressed to set
Description: Sets the value of the attribute *repressed*.

getPiliTimer ()
Type: public int
Inputs: No inputs
Description: Returns the value of the attribute *piliTimer*, which indicates the time required to express the conjugative machinery.

setPiliTimer (int t)
Type: public void
Inputs: - t: the piliTimer to set
Description: Sets the value of the attribute *piliTimer*.

7.1.2 Bacterium

Attributes:

$C0$	
Data type: protected double	**Initial value:** 0
Keeps track of how many iterations the model has been running by storing the value returned by RunEnvironment. getInstance(). getCurrentSchedule(). getTickCount();	

Methods:

Bacterium (double m, State s, double G, boolean R, boolean b)
Type: public
Inputs: - m: The initial cell mass - s: The initial cell state - G: The Generation Time - R: A boolean flag indicating the plasmid has the Tra genes repressed - b: True if plasmid is a mobilizable plasmid
Description: Constructs the cell agent using the constructor of its superclass and assigns the current value of *C0*.

Bacterium (double m, State s, double G, boolean R, boolean b, int pili)
Type: public
Inputs: - m: The initial cell mass - s: The initial cell state - G: The Generation Time - R: A boolean flag indicating the plasmid has the Tra genes repressed - b: True if plasmid is a mobilizable plasmid - *pili*: The pili counter
Description: Constructs the cell agent using the constructor of its superclass, assigns the current value of *C0* and sets the *pili* boolean.

gridUptake(double r)
Type: public double
Inputs: - r: the required calculated uptake value
Description: This method implements the uptake on the agent's current grid cell. Returns the real uptake which depends on nutrient availability on current grid cell. The method evaluates the value layer corresponding to the nutrients in the current location. If the current amount of nutrients in the grid is bigger or equal than the required amount, it returns the required amount. If it is smaller, it returns all the available nutrients in the grid. The grid will contain only the remaining amount of nutrients, if spare.

gridDiffusion(double r)
Type: public double
Inputs: - r: the required calculated uptake
Description: This is an adaptation of the diffusion method as described in the paper: "Modeling the spatial dynamics of plasmid transfer and persistence". Returns the real uptake which depends on nutrient availability on grid. To do so, the methods looks for the amount of nutrients in every moore neighbour at distance one. If it finds a neighbour with any nutrients, the method takes the nutrients in the same way as the previous method and returns it. If not, it will look for that amount of nutrients in all the neighbours at distance 2. If it hasn't still found any nutrient in any grid, it will look for it in the neighbours at distance 3, repeating the process.

UptakeAndGrowth()
Type: public void
Inputs: No inputs
Description: This method implements the Cellular Growth. The method first estimates the required uptake of nutrients based on the cell mass, a linear function or a sigmoideal function (it can be chosen in the code). It then calls the method *gidUptake(reqAmount)* to take nutrients from the grid and store it in the variable v. If the bacterium is a superindividual representing N individuals, reqAmount will be multiplied by N, and the result stored in v will be divided by N. This v has a plasmid penalty in case the bacterium is donor or transconjugant (it is infected). After those calculations, the attribute starved will be true if v is less or equal than 0. If the cell is viable, the method sets its length based in the nutrients. If there are enough nutrients, its mass increases with metabolic efficiency. If there are no enough nutrients, its mass grows with metabolic inefficiency. If the cell is viable, its length will also be modified based on the current length, width, the previous mass and the new mass.

Division()
Type: public void
Inputs: No inputs
Description: Implements the cellular division by creating a variable Zm based on a random variable Z, the average cell mass at division(M) and the standard deviation of cell mass at division($sigmaM$). If the current mass of the cell is bigger than or equal to Zm, or the current length is bigger than the MAX_LENGTH attribute, the cellular division takes place. If there is any empty moore neighbour at distance 1, the new daughter cell is placed in that empty site. Else, adds the new daughter cell to a non-empty site. Daughter cells has the same characteristics as the first cell.

getDaughterCell()
Type: public Bacterium
Inputs: No inputs
Description: Creates a new cell and adjust the parameters of current cell, returning the new cell. The method creates a random number for the plasmid loss probability. If its next double number is bigger than 1 - an already established probability for plasmid loss in the class *MyParameters*, the state of the new cell will be *R*. Else, it will be the state of the originator cell.

addToGrid(Bacterium daughter, int x, int y)
Type: public void
Inputs: - *daughter*: bacterium - *x and y*: coordinates in the grid
Description: This method moves a cell to the given position. An error message is displayed if it is not moved.

PiliExpression()
Type: public void
Inputs: No inputs
Description: This method handle the logic of pili expression for both constitutivelly expressed and repressed plasmids, using a simple probabilistic scheme. If the current state of the cell is R or it is transconjugant but its infected with a mobilizable non-conjugative plasmid, the methods does nothing. The method follows several random operations which can give rise to the pili expression and modifications in the cell mass and length.

isReadyForConjugate()
Type: public boolean
Inputs: No inputs
Description: Returns a decision rule for conjugation using a random variable for cell division.

Conjugation()
Type: public void
Inputs: No inputs
Description: This method takes place only if the cell state is Donor or if it is Transconjugant and is infected by a mobilizable and conjugative plasmid. Altough those two conditions are true, the cell will not conjugate if the random method isReadyForConjugate () returns false. It evaluates the probability of mating-pair formation as function of pili number, and starts looking for a neighbour to transfer the plasmid. When the plasmid is transferred, its mass decreases.

Shoving()
Type: public void
Inputs: No inputs
Description: Shoving relaxation process. Adapted from "BacSim, a simulator for individual-based modelling of bacterial colony growth". The method initialises a shoving relaxation vector and performs the following actions for each moore neighbour at distance 1 from the current cell: The shoving vector is calculated, and the cell is moved according to this vector. If the new calculated position is at the outer 25% of the grid cell, the agent is moved to the cell directly adjacent.

step()
Type: public void
Inputs: No inputs
Description: This is a scheduled method to be performed in each interval of time. If the cell is viable, it calculates its shoving forces, the uptake of nutrients and growth, the division, the pili expression and the conjugation processes in this order.

7.1.3 BacteriumEquations

Attributes:

No attributes.

Methods:

eqnEstimateZm (double M, double sigmaM, double Z)
Type: public static double
Inputs: - M: The average cell division mass - $sigmaM$: The standard deviation of cell division mass - Z: Sampled random variable from a standard normal distribution
Description: Returns the possible value of division mass.

eqnAverageUptake(double G, double m0, double Zm, double e)
Type: public static double
Inputs: - *G*: The generation time - *m0*: The mass at t=0 - *e*: The metabolic efficiency
Description: Calculate the average nutrient based on the estimated cell duration. This is a simple linear fit based on the desired generation time parameter. Returns the average nutrient uptake.

eqnUptake(double mass, double V, double Zm)
Type: public static double
Inputs: - *mass*: The current cell mass - *V*: The average nutrient uptake - *Zm*: The estimated mass at cell should divide
Description: Calculate the uptake as function of current cell mass. Returns the required uptake.

eqnLinearUptake(double mass, double m0, double V, double Zm)
Type: public static double
Inputs: - *mass*: The current cell mass - *m0*: The initial cell mass - *V*: The average nutrient uptake - *Zm*: The estimated mass at cell should divide
Description: Calculate the linear uptake as function of current cell mass. Returns the required uptake.

eqnSigmoidUptake(double mass, double m0, double V, double Zm)
Type: public static double
Inputs: - *mass*: The current cell mass - *m0*: The initial cell mass - *V*: The average nutrient uptake - *Zm*: The estimated mass at cell should divide
Description: This is a simple case of a logistic sigmoid uptake function. Returns the required uptake.

eqnLength(double l0, double w0, double w1, double m0, double m1)
Type: public static double
Inputs: - *l0*: current length - *w0*: current width - *w1*: new width (in rod shaped bacteria it is aproximatelly constant (0.5 micrometers) - *m0*: previous cell mass - *m1*: new cell mass
Description: Calculate the increment in the cell length as function of cell mass. Returns the new calculated cell length.

eqnGrowth(double m, double v, double e)
Type: public static double
Inputs: - *m*: Current mass - *v*: Effective uptake - *e*: Metabolic efficiency
Description: Calculate how the cell mass should increase when nutrient are available. Returns the new cellular mass.

eqnDecay(double m, double v, double e)
Type: public static double
Inputs: - *m*: Current mass - *v*: Effective uptake - *e*: Metabolic efficiency
Description: Calculates how the cell mass should decrease when nutrient are unavailable. Returns the new cellular mass.

eqnConjugationProbability(double pili, double maxpili)
Type: public static double
Inputs: - *pili*: actual number of t4ss pili on cell surface - *maxpili*: average maximal number of conjugative pili on e. coli cells
Description: Calculates the probability p of mate pair formation. Returns the probability of successful conjugative events.

FgToKg(double fg)
Type: public static double
Inputs: - *fg*: femtogram value
Description: Converts femtograms to kilograms. Returns the corresponding kg value.

KgToFg(double kg)
Type: public static double
Inputs: - *kg*: kilogram value
Description: Converts kilograms to femtograms. Returns the corresponding ft value.

MuToMeters(double mu)
Type: public static double
Inputs: - *mu*: The value in micrometers
Description: Micrometers to meters conversion method. Returns the same value in meters.

cellVolume(double w, double l)
Type: public static double
Inputs: - w: cellular width - l: cellular length
Description: Calculates the cell volume based in the width and the length. Returns the volume in m^3.

cellDensityFromWetMass(double m, double w, double l)
Type: public static double
Inputs: - m: cellular mass - w: cellular width - l: cellular length
Description: Calculates the cell density based on the cell volume and mass. Returns the density in kg/m^3.

cellLengthFromWetMass(double m, double d)
Type: public static double
Inputs: - m: cellular mass - d: cellular density
Description: Calculates the cell length based on its mass and density. Returns the length in Mu.

7.1.4 BacteriumStyle2D

Attributes:

Attribute	**Data type**	**Initial value**
shapeFactory	private ShapeFactory2D	No initialised
Description: This object can create different spatial figures with the given dimensions.		

Methods:

init(ShapeFactory2D factory)
Type: public void
Inputs: - *factory*: Object which can create different spatial figures with the given dimensions
Description: Constructor which nitialises the attribute *shapeFactory*.

getColor(Object agent)
Type: public Color
Inputs: - *agent*: Object representing any agent which it is wanted to know its *state*
Description: Given an agent as input, this method returns its corresponding colour depending on its state: Donor = dark red R = dark green T = dark blue

getVSpatial(Object agent, VSpatial spatial)
Type: public VSpatial
Inputs: - *agent*: Object representing any agent - *spatial*: The object which contains the shape of the agent
Description: This method returns the value of the input *spatial*. If such input is null, the method assigns a circular shape to it.

7.1.5 ModelRatesHelper

Attributes:

Attribute	**Data type**	**Initial value**
$pN0$	private double	0
Description: Initial population.		

Attribute	**Data type**	**Initial value**
D	private double	0
Description: Current donor population.		

Attribute	**Data type**	**Initial value**
R	private double	0
Description: The recipient population.		

Attribute	**Data type**	**Initial value**
T	private double	0
Description: The transconjugant population.		

Attribute	**Data type**	**Initial value**
Cd	private double	0
Description: Conjugative transfers performed by Donor cells.		

Attribute	**Data type**	**Initial value**
Ct	private double	0
Description: Conjugative transfers performed by Transconjugant cells.		

Attribute	**Data type**	**Initial value**
instance	private final static ModelRatesHelper	new ModelRatesHelper()
Description: Private static attribute which is an object of the class itself.		

Methods:

ModelRatesHelper()
Type: Private
Inputs: No inputs
Description: Private constructor. No instances can be created outside this class.

getGammaEndPointAll(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc (it extends the class $Collection < T >$)
Description: Returns the gamma end point of all the population.

getGammaEndPointDonor(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: Returns the gamma end point of donors in the population.

getGammaEndPointTransconjugant(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: Returns the gamma end point of transconjugants in the population.

getRCv(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: If $(v = T - (Cd + Ct)/T)! = NaN$, it returns v. Else, it returns 0.

getRCh(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: If $(v = (Cd + Ct)/T)! = NaN$, it returns v. Else, it returns 0.

getRCt(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: If $(v = Ct/(Cd + Ct))! = NaN$, it returns v. Else, it returns 0.

getTransconjugantPerRecipientCell(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: Returns $T/(R + T)$.

getGrowthRateAll(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: Returns the growth rate of all the population.

getGrowthRateDonors(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: Returns the growth rate of donors of the population.

getGrowthRateRecipient(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: Returns the growth rate of recipient cells of the population.

getGrowthRateTransconjugant(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: Returns the growth rate of transconjugants of the population.

getGenerationTimeAll(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: Returns the generation time of all the population.

getGenerationTimeDonors(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: Returns the generation time of donors of the population.

getGenerationTimeRecipient(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: Returns the generation time of recipient cells of the population.

getGenerationTimeTransconjugant(Context context)
Type: public double
Inputs: - *context*: A set of agents, projections, layers, etc
Description: Returns the generation time of transconjugants of the population.

getGrowthRate(double N, double N0, double t, double t0)
Type: public double
Inputs: - N: The current population - *N0*: The initial population - t: Input which keeps track of how many iterations the model has been running - *t0*: How many iterations the model has been running at the instant 0
Description: Given the number of individuals in a population (D, R, T or all of them) calculates the growth rate of such population.

getGenerationTimes(double N, double N0, double t, double t0)
Type: public double
Inputs: - N: The current population - *N0*: The initial population - t: Input which keeps track of how many iterations the model has been running - *t0*: How many iterations the model has been running at the instant 0
Description: Returns the generation time of a given population N.

gammaEndPoint(double psi, double N, double N0, double D, double R, double T)
Type: public double
Inputs: - *psi*: The growth rate - *N*: The current population - *N0*: The initial population - *D*: The current donor population - *R*: The recipient population - *T*: The transconjugant population
Description: Estimates the conjugation rate using the end-point method. Returns the gama value ($mlcell^{-1}h^{-1}$).

ModelRatesHelper getInstance()
Type: public static
Inputs: No inputs
Description: Returns the private static attribute *instance*, as part of the singleton pattern underlying this class.

gatherData()
Type: public void
Inputs: No inputs
Description: Method which initialises the attributes pN0, D, R, T, Cd, Ct taking their values from the attributes of the class MyPopulationBookkeeper.

7.1.6 MyContextBuilder

Attributes:

width	
Data type: private int	**Initial value:** No initialised
The desired grid width. Obtained from the class MyParameters.	

height	
Data type: private int	**Initial value:** No initialised
The desired grid height. Obtained from the class MyParameters.	

Methods:

build(Context<Object> context)
Type: public Context
Inputs: - *context*: A set of agents, projections, layers, etc
Description: This is the main method of the class. It configures the simulation parameters and adds agents and projections to the context, among other information. To do so, it initialises a context and sets a GridBuilderParameters object for creating a multi occupancy 2 dimensional grid. When created, it uses both the context and the parameters as inputs for the method GridFactory.createGrid(...) which returns a grid with the desired properties. It then initialises the random streams, the population and the nutrients. It also sets an adder into the grid which can be used to add more objects to the grid. Returns the initialised context.

configSimulation()
Type: private void
Inputs: No inputs
Description: If the simulation is running in batch mode, it sets the simulation time to 8 hours. Else, it sets the simulation time to 10 hours.

initRandomStreams()
Type: private void
Inputs: No inputs
Description: This method sets a random seed an several probabilistic distributions in the class RandomHelper: - Cellular division = normal distribution - Neighbourhood, plasmid loss, conjugation and pilus = Uniform distribution - Repressed and derepressed = poison distribution

initPopulation(Context<Object> context, Grid grid)
Type: private void
Inputs: - *context*: A set of agents, projections, layers, etc - *grid*: Projection object of type grid (included in the context as another projection)
Description: Creates the initial population of D and R cells and distribute it randomly across the grid. - Resets the instance of MyPopulationBookkeeper - Gets the number of individuals based on the method MyParameters.getN0(), which returns the number of individuals based on the established width and height of the grid. - Initialises the population, establishing the number of donor and recipient bacteria, and adds them to the *context*. - Distributes each bacterium in the *grid* projection randomly.

initNutrients(Context<Object> context)
Type: private void
Inputs: - *context*: A set of agents, projections, layers, etc
Description: This method initialize the nutrient on the grid value layer. To do so, the method adds a nutrient *ValueLayer* to the context, and sets the amount of nutrients in each point of such layer.

7.1.7 MyNeighborhood

Attributes:

No attributes.

Methods:

getMooreNeighborhood(GridPoint pt, int size, boolean c)
Type: public static List<GridPoint>
Inputs: - *pt*: GridPoint where the agent is placed - *size*: Size of the moore neigborhood (1, 2 or 3) - *c*: *true* if it takes into account the cell of the agent calling this method
Description: Returns a shuffled list of the GridPoints corresponding to the moore neighbors of *pt* in a moore neighborhood of *size*.

getMooreNeighborhood(Object o, int size, boolean c)
Type: public static List<GridPoint>
Inputs: - *o*: It is usually an agent of whose neighbor GridPoints have to be returned - *size*: Size of the moore neigborhood (1, 2 or 3) - *c*: *true* if it takes into account the cell of the agent calling this method
Description: Returns a shuffled list of the GridPoints corresponding to the moore neighbors of the agent *o* in a moore neighborhood of *size*. This method does not need a GridPoint as input to get the list of neighbors as it can use the agent *o* to get its GridPoint and call the previous method as return statement.

getMooreNeighborhood(Object o, GridPoint pt, int size)
Type: public static List<GridPoint>
Inputs: - *o*: It is usually an agent of whose neighbor GridPoints have to be returned - *pt*: GridPoint where the agent is placed. - *size*: Size of the moore neigborhood (1, 2 or 3)
Description: Returns a shuffled list of the GridPoints corresponding to the moore neighbors of the GridPoint *pt* in a moore neighborhood of *size*. This method is similar to the first one. The only difference is this method does not use the boolean *c*.

getEmptyMooreNeighborhood(Object o, GridPoint pt, int size)
Type: public static List<GridPoint>
Inputs: - *o*: It is usually an agent of whose empty neighbor GridPoints have to be returned - *pt*: GridPoint where the agent is placed - *size*: Size of the moore neigborhood (1, 2 or 3)
Description: Returns a shuffled list of the empty GridPoint neighbors of the agent *o*.

getEmptyMooreNeighborhood(Object o, int size)
Type: public static List<GridPoint>
Inputs: - *o*: It is usually an agent of whose empty neighbor GridPoints have to be returned - *size*: Size of the moore neigborhood (1, 2 or 3)
Description: Returns the empty GridPoint neighbors of the agent *o*. Same method as the previous one, but it does not need the GridPoint *pt* of the agent *o* as input.

ListFromIterator(Iterator i)
Type: public static List
Inputs: - *i*: Iterator object which contains individuals
Description: This method returns an object of the class List with the individuals contained in an Iterator object *i*. The returned List object is shuffled following an uniform random distribution based on MyParameters.*RANDOM_NEIGHBORHOOD*.

getUnitVector(Object o, GridPoint pt, GridPoint other)
Type: public static double[]
Inputs: - *o*: Object needed to obtain the grid in which it is contained - *pt*: GridPoint A - *other*: GridPoint B
Description: Returns the unit vector of the GridPoints *pt* and *other*.

7.1.8 MyParameters

N_SCALEFACTOR	
Data type: public final static int	**Initial value:** 100
Super-individual scale-factor.	

CONTEXT	
Data type: public static String	**Initial value:** "ConjugationKinetics-1"
Context of the simulation.	

GRID	
Data type: public static String	**Initial value:** "Grid"
Name of the grid used in the simulation.	

RANDOM_DIVISION	
Data type: public static String	**Initial value:** "DivisionMassNormal"
Name of the normal distribution for the cellular division.	

RANDOM_NEIGHBORHOOD	
Data type: public static String	**Initial value:** "NeighborhoodUniform"
Name of the uniform distribution for choosing random neighbors.	

RANDOM_PLASMIDLOSS	
Data type: public static String	**Initial value:** "PlasmidLossUniform"
Name of the uniform distribution for the plasmid loss.	

RANDOM_CONJUGATION	
Data type: public static String	**Initial value:** "ConjugationUniform"
Name of the uniform distribution for the conjugation process.	

RANDOM_PILUS	
Data type: public static String	**Initial value:** "PilusExpression"
Name of the random distribution of pilus expression.	

RANDOM_REPRESSED	
Data type: public static String	**Initial value:** "Repressed"
Name of the random distribution of repression of plasmids.	

RANDOM_DEREPRESSED	
Data type: public static String	**Initial value:** "DeRepressed"
Name of the random distribution of derepression of plasmids.	

RANDOM_PLASMIDLOSS	
Data type: public static String	**Initial value:** "PlasmidLossUniform"
Name of the uniform distribution of plasmid loss.	

VL_NUTRIENTS	
Data type: public static String	**Initial value:** "Nutrients"
Value Layer nutrient key.	

RANDOM_SEED	
Data type: public static String	**Initial value:** "randomSeed"
Name of the random seed.	

PENALTY_PLASMID	
Data type: public static String	**Initial value:** "fixed"
Metabolic burden by plasmid harbouring.	

PENALTY_CONJUGATION	
Data type: public static String	**Initial value:** "conjugation"
Metabolic burden by conjugation event.	

RANDOM_PLASMIDLOSS	
Data type: public static String	**Initial value:** "PlasmidLossUniform"
Name of the uniform distribution for the plasmid loss.	

PENALTY_DEREPRESSION	
Data type: public static String	**Initial value:** "derepression"
Metabolic burden by T4SS expression.	

TAU	
Data type: public static String	**Initial value:** "GT"
Generation time.	

ORIT	
Data type: public static String	**Initial value:** "orit"
Self-transmissible plasmid, oriT=true.	

MIN_CELL_CYCLE	
Data type: public static String	**Initial value:** "CC"
Minimal cell cycle to conjugate.	

DONOR_DENSITY	
Data type: public static String	**Initial value:** "donors"
The initial density of donor cells.	

NUTRIENT	
Data type: public static String	**Initial value:** "nutrient"
The initial nutrient particles.	

REPRESSED	
Data type: public static String	**Initial value:** "repressed"
Name of repressed attribute.	

RECOVERY_TIME	
Data type: public static String	**Initial value:** "recovery"
Name of recovery time attribute.	

MATURATION_TIME	
Data type: public static String	**Initial value:** "maturation"
Name of the attribute maturation.	

PLASMID_LOSS	
Data type: public static String	**Initial value:** "plasmidloss"
Name of the plasmid loss.	

GRID_HEIGHT	
Data type: public static String	**Initial value:** "height"
The desired grid height.	

GRID_WIDTH	
Data type: public static String	**Initial value:** "width"
The desired grid width.	

SI_MICROMETERS	
Data type: public static double	**Initial value:** 1.0000e-006
Micrometers units.	

SI_FEMTOGRAMS	
Data type: public static double	**Initial value:** 1.00e-018
Femtograms units.	

parm	
Data type: private final static Parameters	**Initial value:** No initialised (see ***static*** block in methods)
Encapsulates simulation time model parameters.	

Methods:

static
parm = RunEnvironment.getInstance().getParameters();
Description:
Static initializer. It is executed when the class is loaded (or initialized, to be precise).

getRandomSeed()
Type: public static int
Inputs:
No inputs
Description:
Returns the simulation random seed.

getPlasmidPenalty()
Type: public static double
Inputs: No inputs
Description: Returns the *plasmidPenalty*.

getConjugationPenalty()
Type: public static double
Inputs: No inputs
Description: Returns the *conjugationPenalty*.

getDerepressionPenalty()
Type: public static double
Inputs: No inputs
Description: Returns the derepressionPenalty.

getGenerationTime()
Type: public static int
Inputs: No inputs
Description: Returns the *generationTime*.

getMinCellCycle()
Type: public static double
Inputs: No inputs
Description: Returns the *minCellCycle*.

getDonorDensity()
Type: public static int
Inputs: No inputs
Description: Returns the *donorDensity*.

isOriT()
Type: public static boolean
Inputs: No inputs
Description: Returns the oriT.

getNutrient()
Type: public static double
Inputs: No inputs
Description: Returns the intial nutrient concentration.

isRepressed()
Type: public static boolean
Inputs: No inputs
Description: Returns if plasmid is constitutively expressed.

getPlasmidRecoveryTime()
Type: public static double
Inputs: No inputs
Description: Returns the plasmid recovery time.

getPlasmidMaturationTime()
Type: public static double
Inputs: No inputs
Description: Returns the plasmid maturation time.

getPlasmidLossProbability()
Type: public static double
Inputs: No inputs
Description: Returns the plasmid loss probability.

getHeight()
Type: public static int
Inputs: No inputs
Description: Returns the height.

getWidth()
Type: public static int
Inputs: No inputs
Description: Returns the width.

getN0()
Type: public static int
Inputs: No inputs
Description: Returns the initial population based in the formula $N0 = \frac{\sqrt{height * width}}{20}$.

7.1.9 MyPopulationBookkeeper

Attributes:

D	
Data type: private AtomicInteger	**Initial value:** No initialised
Number of donor cells in the population.	

R	
Data type: private AtomicInteger	**Initial value:** No initialised
Number of recipent cells in the population.	

T	
Data type: private AtomicInteger	**Initial value:** No initialised
Number of transconjugant cells in the population.	

Cd	
Data type: private AtomicInteger	**Initial value:** No initialised
Number of conjugative transfers performed by Donor cells.	

Ct	
Data type: private AtomicInteger	**Initial value:** No initialised
Number of conjugative transfers performed by Transconjugant cells.	

instance	
Data type: private static MyPopulationBookkeeper	**Initial value:** new MyPopulationBookkeeper()
Private and static instance of the class itself to implement the singleton pattern, so only one instance of the object can be created.	

Methods:

MyPopulationBookkeeper()
Type: private
Inputs: No inputs
Description: Initialises the variables D, C, T, Cd and Ct with the value "new AtomicInteger(0)", an int value which can be updated automatically.

getInstance()
Type: public static MyPopulationBookkeeper
Inputs: No inputs
Description: Public method to return the only instance of the class itself. This method is part of the singleton pattern applied to this class.

reset()
Type: public void
Inputs: No inputs
Description: Sets the value of D, C, T, Cd and Ct to 0.

getD()
Type: public double
Inputs: No inputs
Description: Returns D * MyParameters.N_SCALEFACTOR (the number of Donor cells * Super-individual scale-factor, which is 100).

incrementD()
Type: public void
Inputs: No inputs
Description: Increments by one the current value of D.

decrementD()
Type: public void
Inputs: No inputs
Description: Decrements by one the current value of D.

getR()
Type: public double
Inputs: No inputs
Description: Returns R * MyParameters.N_SCALEFACTOR (the number of Recipent cells * Super-individual scale-factor, which is 100).

incrementR()
Type: public void
Inputs: No inputs
Description: Increments by one the current value of R.

decrementR()
Type: public void
Inputs: No inputs
Description: Decrements by one the current value of R.

getT()
Type: public double
Inputs: No inputs
Description: Returns T * MyParameters.N_SCALEFACTOR (the number of transconjugant cells * Super-individual scale-factor, which is 100).

incrementT()
Type: public void
Inputs: No inputs
Description: Increments by one the current value of T.

decrementT()
Type: public void
Inputs: No inputs
Description: Decrements by one the current value of T.

getN0()
Type: public double
Inputs: No inputs
Description: Returns MyParameters.getN0() * MyParameters.N_SCALEFACTOR(initial population * Super-individual scale-factor, which is 100).

getAll()
Type: public double
Inputs: No inputs
Description: Returns $D + R + T$.

getCd()
Type: public double
Inputs: No inputs
Description: Returns *Cd* * MyParameters.N_SCALEFACTOR (the number of conjugative transfers performed by donor cells * Super-individual scale-factor, which is 100).

incrementCd()
Type: public void
Inputs: No inputs
Description: Increments by one the current value of Cd.

decrementCd()
Type: public void
Inputs: No inputs
Description: Decrements by one the current value of Cd.

getCt()
Type: public double
Inputs: No inputs
Description: Returns *Ct* * MyParameters.N_SCALEFACTOR (the number of conjugative transfers performed by transconjugant cells * Super-individual scale-factor, which is 100).

incrementCt()
Type: public void
Inputs: No inputs
Description: Increments by one the current value of Ct.

decrementCt()
Type: public void
Inputs: No inputs
Description: Decrements by one the current value of Ct.

Bibliography

[1] Etc Group, ETC Group, et al. Extreme genetic engineering: An introduction to synthetic biology. *Ottawa: ETC Group*, pages 1–64, 2007.

[2] Priscilla EM Purnick and Ron Weiss. The second wave of synthetic biology: from modules to systems. *Nature Reviews Molecular Cell Biology*, 10(6):410–422, 2009.

[3] Ron Weiss, Subhayu Basu, Sara Hooshangi, Abigail Kalmbach, David Karig, Rishabh Mehreja, and Ilka Netravali. Genetic circuit building blocks for cellular computation, communications, and signal processing. *Natural Computing*, 2(1): 47–84, 2003.

[4] Jordan Baumgardner, Karen Acker, Oyinade Adefuye, Samuel Thomas Crowley, Will DeLoache, James O Dickson, Lane Heard, Andrew T Martens, Nickolaus Morton, Michelle Ritter, et al. Solving a hamiltonian path problem with a bacterial computer. *Journal of biological engineering*, 3(1):11, 2009.

[5] Angel Goni Moreno. Communication architectures for algorithmic computing in multi-strain bacterial communities. 2010.

[6] Alvin Tamsir, Jeffrey J Tabor, and Christopher A Voigt. Robust multicellular computing using genetically encoded nor gates and chemical wires. *Nature*, 469 (7329):212–215, 2010.

[7] Jeffrey J Tabor, Howard Salis, Zachary B Simpson, Aaron A Chevalier, Anselm Levskaya, Edward M Marcotte, Christopher A Voigt, and Andrew D Ellington. A synthetic genetic edge detection program. *Cell*, 137(7):1272, 2009.

[8] Antonio Prestes Garcia. *A first approach to individual-based modeling of the bacterial conjugation dynamics.* PhD thesis, Informatica, 2011.

[9] Jasna Rakonjac, Nicholas J Bennett, Julian Spagnuolo, Dragana Gagic, and Marjorie Russel. Filamentous bacteriophage: biology, phage display and nanotechnology applications. *Current issues in molecular biology*, 13(2):51, 2011.

[10] Marjorie Russel, Henry B Lowman, and Tim Clackson. Introduction to phage biology and phage display. *Practical Approach to Phage Display*, pages 1–26, 2004.

[11] L Chasteen, J Ayriss, P Pavlik, and ARM Bradbury. Eliminating helper phage from phage display. *Nucleic acids research*, 34(21):e145–e145, 2006.

[12] Yi Sun and Liang Cheng. A survey on agent-based modelling and equation-based modelling. *Department of Computer Science. Georgia State University. Atlanta.*

[13] Volker Grimm, Uta Berger, Donald L DeAngelis, J Gary Polhill, Jarl Giske, and Steven F Railsback. The odd protocol: A review and first update. *Ecological Modelling*, 221(23):2760–2768, 2010.

[14] F.. Hellweger and V. Bucci. A bunch of tiny individuals. individual-based modeling for microbes. *Elsevier*, 2009.

[15] A.Goni and M. Amos. Discrete modelling of bacterial conjugation dynamics. November 2012.

[16] R. Egbert S. Jang, K. Oishi and E. Klavins. Specification and simulation of synthetic multicelled behaviors. *ACS Synthetic Biology.*

[17] Andrew Phillips Timothy J. Rudge, Paul J. Steiner and Jim Haseloff. Computational modeling of synthetic microbial biofilms. *ACS Synthetic Biology.*

[18] Sonia Martins Andreas Dtsch Cristian Picioreanu Jan-Ulrich Kreft Laurent A. Lardon, Brian V. Merkey and Barth F. Smets. idynomics: next-generation individual-based modelling of biofilms. *Environmental Microbiology*, 2011.

[19] Kevin M Esvelt, Jacob C Carlson, and David R Liu. A system for the continuous directed evolution of biomolecules. *Nature*, 472(7344):499–503, 2011.

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Fri Feb 14 20:17:41 CET 2014
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)